USING THE 65C02 ASSEMBLER
(Version 1.60)

This guide describes how to use the 65C02 Assembler produced by Alan Phillips of the Department of Computing at Lancaster University. The reader is assumed to be familiar with the concepts of programming at assembly level.

The 65C02 Assembler is copyright (C) Alan Phillips 1987. It may be passed on by anyone, to anyone, and used for any peaceful purpose whether private or commercial. No purchase, licensing or permission is needed. It may be distributed in any way or posted on any Bulletin Board system provided that it is not sold (apart from reasonable handling and media costs), that due credit is made for authorship, and that this paragraph is brought to the attention of the recipient.

Edition 3.1
October 1987
Alan Phillips

CONTENTS

1. INTRODUCTION
    1.1 Installing the Assembler
    1.2 The Assembler's CLI mode
    1.3 Creating source files
    1.4 Starting an assembly
        1.4.1 The command line options
        1.4.2 Alternative syntax forms for the command line
    1.5 Assembly on a second processor
    1.6 The object code buffer
    1.7 Returning to the calling language
    1.8 Use of control keys during assembly

2. SOURCE PROGRAM FORMAT
    2.1 Syntax of the line
        2.1.1 The label field
        2.1.2 The opcode field
        2.1.3 The operand field
        2.1.4 The comment field
    2.2 The syntax of expressions
        2.2.1 Numeric expressions
        2.2.2 Strings
        2.2.3 String expressions

3. THE OUTPUT LISTING
    3.1 The source listing level
    3.2 Source listing format
        3.2.1 Listing object code bytes
    3.4 Symbol table listing format
    3.5 Error reports
    3.6 Output page control
        3.6.1 Setting page dimensions
        3.6.2 Setting the page title
        3.6.3 Setting the timestamp
        3.6.4 Page layout directives
    3.7 Controlling the output destination
        3.7.1 Output to a printer
        3.7.2 Output to a file
        3.7.3 Printer control

4. DIRECTIVES DEFINING DATA AND CONSTANTS
    4.1 Defining constants
    4.2 Byte and word directives
        4.2.1 Byte directives
        4.2.2 Word directives
        4.2.3 Using repeat counts
    4.3 Character string directives
        4.3.1 The ASC and STR directives
        4.3.2 The CASC and CSTR directives
        4.3.3 Planting special characters
    4.4 The HEX directive
    4.5 The CODE directive

5. OBJECT FILE CONTROL DIRECTIVES
    5.1 Defining the current address
    5.2 Using dummy sections
    5.3 Reading the current address
        5.3.1 The DS directive
        5.3.2 Reading the current address
    5.4 Defining object load and execution addresses
        5.4.1 Specifying the low bytes of the addresses

6. SOURCE FILE CONTROL DIRECTIVES
    6.1 Chaining source files
    6.2 Including one file into another

7. CONDITIONAL ASSEMBLY
    7.1 Assembly conditional on expressions
    7.2 Assembly conditional on the existence of symbols
    7.3 Nesting conditionals
    7.4 Listing conditionals

8. PROGRESS REPORTING DIRECTIVES
    8.1 The DISP, DISP1 and DISP2 directives
    8.2 The WAIT, WAIT1 and WAIT2 directives
    8.3 The QUERY directive
    8.4 The STOP directive

9. WRITING SIMPLE MACROS
    9.1 Using macro parameters
    9.2 Specifying macro parameters
    9.3 Nesting macros
    9.4 Redefining opcodes and directives
    9.5 Labels within macros

10. THE MACRO PROGRAMMING LANGUAGE
    10.1 Sequence Symbols
    10.2 Assembly Time Variables
        10.2.1 Creating Assembly Time Variables
            10.2.1.1 Local and global ATVs
            10.2.1.2 String and numeric values
            10.2.1.3 Efficient use of memory
        10.2.2 Simple substitution of Assembly Time Variables
    10.3 Writing complex macros
        10.3.1 Programming macro loops
            10.3.1.1 Loops controlled by counter
            10.3.1.2 Loops accessing macro parameters
        10.3.2 Changing macro parameters
        10.3.3 Listing control for macros
        10.3.4 Exiting a macro prematurely
        10.3.5 Comment lines within macros
    10.4 System ATVs

Appendices

A1. Opcodes and addressing modes
A2. Assembler directives
A3. Differences from the ADE Assembler
A4. Differences from previous releases
A5. Customising the Assembler

Acknowledgments

Thanks are due to Dave Morriss, Neil Mercer, Alan Baker, Peter Vince, Mike Tubby, Scott Larson and Peter Francis for many helpful suggestions and comments during the development of the Assembler.

USING THE 65C02 ASSEMBLER

1. INTRODUCTION

The 65C02 Assembler runs in sideways RAM or as a sideways ROM on the BBC Models B, B+, B+128, Master 128 and Master Compact, with or without a 6502 Second Processor. It supports all the opcodes of the 6502 and 65C02 processor families, and contains a powerful Macro Programming Language.

The Assembler is a disc-oriented system. Source files must be held on disc, and the object code will be written to disc. It can be used with the Acorn DFS, ADFS, NFS and any other Acorn-compatible disc or network filing system, but not on cassette-only machines.

There are no source editing facilities contained in the Assembler. However, it is able to accept source produced by most text editors or word-processor programs.

1.1 Installing the Assembler

The 65C02 Assembler code is completely unprotected. If you have a machine with sideways RAM, you will be able to load it to a RAM bank and run it; if not, you will need to program it into a 27128 EPROM and fit it into a sideways ROM socket.

1.2 The Assembler's CLI mode

If you have the Assembler fitted into the highest priority ROM slot of your machine, pressing CONTROL-BREAK or powering on will enter it as the current language. In this case the Assembler is said to be in "CLI (Command Language Interpreter) mode". You will see a "*" prompting you for input, and any lines you type will be sent to the BBC Operating System as MOS commands. In this mode there is no need to type a "*" yourself in front of the commands, although it won't matter if you do.

You can also enter CLI mode by issuing the command

```
*CLI
```

regardless of the ROM socket the Assembler is in.

From CLI mode you can of course start an assembly with the

*ASSEMBLE command described below. Alternatively, you could enter another language by typing, say,

*BASIC

or

*WORDWISE

Naturally, you are able to issue DFS commands in CLI mode, or you could *TYPE files, and so on. You can't use the BASIC statement MODE to change screen mode, though, since the current language is not BASIC. However, for convenience the Assembler will respond to a command *MODE, so you could type

*MODE 3

to set mode 3 if you wished. This command is available either from the Assembler's CLI mode or from any other language.

1.3 Creating source files

Though the Assembler does not contain any editing facilities, it will accept files produced on just about any word processor or source editor. The source you create should contain only printable characters, spaces and TAB characters, and each line should end with a carriage return byte (with optionally a line feed character before or after it). The Assembler will treat the character used in saved Wordwise files to indicate a TAB (ASCII code &DD) as a "real" TAB character.

The Assembler will normally ignore any other characters in the source. Some word processors, though, may use characters with ASCII codes in the range 128 to 255 for special purposes: the Assembler can be configured to change its action on encountering these. This is described in detail in Appendix 5.

You can create suitable files easily with WordWise or View:

a. Using WordWise

Files saved by menu option 1 can be used directly as input to the Assembler, provided that you have not included any embedded commands. The WordWise TAB character, which appears on the editing screen as a right-pointing arrow, will be accepted as a TAB by the Assembler. Remember to press RETURN at the end of each source line.

Alternatively, you can spool the file to disc first using menu

option 8 - but this will create a bigger file, that will take longer to assemble. You may need some caution here if you are using the "|" character within string constants: remember that this is the WordWise "pad" character, so by default will appear as a space in the spooled file. To overcome this, use the WordWise "pc" directive to redefine the pad character to something else.

b. Using View

A file created by the View SAVE, WRITE or EDIT commands can be directly input to the Assembler. You should not include rulers, stored commands or highlight codes in this file. You should always remember to press RETURN at the end of every source line: it will be necessary to turn off the Format and Justify options when typing the source, otherwise View will interfere with this.

1.4 Starting an assembly

The Assembler is started with the *ASSEMBLE command. This is followed by a number of "option flags" that tell the Assembler exactly how you wish it to operate.

The simplest use of the command would be, for example:

```
*ASSEMBLE SOURCE
```

where the source file name is SOURCE. This format performs a "syntax check run" - no object code is produced. This will give you the fastest possibly assembly, and it can be very useful when checking a large program for errors. To produce an object file containing the assembled code, you need to specify a "-O" option:

```
*ASSEMBLE SOURCE -OPROGRAM
```

where the "-O" is followed by the name of the object file you wish to produce.

After assembly, the Assembler will return you to the language you were running when you issued the *ASSEMBLE command. If you have a machine with shadow screen capability, the Assembler will automatically turn it on to give you the maximum space for symbols. It will return the shadow screen to its initial state when it finishes, and since this involves clearing the screen in a mode change, it will pause for you to press a key before it does so.

1.4.1 The command line options

The various options you can specify to the command are listed below. Each option flag consists of a "-" character followed by a letter, for example "-S". Some option flags must be followed immediately by numbers or filenames. You must separate one option from another by spaces.

Options can be given in any order, but the first parameter to the command must always be the name of the first (or only) source file.

-A You must specify this option to assemble the source for BBC KERMIT, and it is recommended for sources originally written for the ADE Assembler. It makes the syntax of labels ADE-compatible, and treats only the first 6 characters of each as significant. This option implies a -R.

   By default, all the characters of a label are significant.

-B Specifies that a memory buffer is to be used for object code. This must be followed by a number in the range 1..16 (see section 1.6)

   By default, no object buffer is used, and bytes are written to the object file one by one.

-C Specifies the default code listing level (see the CLST directive). This must be followed by 0, 1 or 2.

   By default, the Assembler will start an assembly with a code listing level of 1.

-F Specifies the name of a listing file, which must follow the option flag. If you give this option, the assembly listing will be written to the file instead of being sent to the screen.

   If you do not use either the -P or -F options, the listing will be sent to the screen only.

   Note that if you specify both -P and -F options, only the -F will take effect.

-G Specifies that the Assembler is to restart the calling language immediately the assembly is over. (See section 1.7)

By default, the Assembler pauses after assembly unless it is called from BASIC or its own CLI mode.

-L Specifies the default source listing level (see the LST directive). This must be followed by 0, 1, 2 or 3. You can also control the list level during the actual assembly with function keys f0, f1, f2 and f3.

By default, the Assembler will start an assembly with a source listing level of 0.

-M Instructs the Assembler to change screen mode before it starts to process the source. The number of the mode you wish to use should follow the letter M, and must be

in the range 0..7.

By default, the Assembler will leave the screen mode unchanged.

-O Specifies the name of the object file, which must follow the option flag. If you omit this, the Assembler performs a syntax check only and does not produce an object file.

-P Specifies that the listing is to be sent to a printer only.

If you do not use either the -P or -F options, the listing will be sent to the screen only.

Note that if you specify both -P and -F options, only the -F will take effect.

-R If this is specified, 65C02 opcodes and addressing modes are rejected. If you omit it, all 65C02 codes are assembled.

-S Specifies that lines skipped in conditionals are not listed (see the SFCOND and LFCOND directives)

By default, the Assembler will list all skipped lines in conditionals.

-W Specifies that the Assembler should wait after displaying a line containing an error, so you can note down the details. To resume assembly, press any key. This option has no effect if you are sending output to a printer or to a file. You can change the setting of this option at any time during assembly by pressing CONTROL-W.

By default, the Assembler will report all errors, but

will not pause.

As an example, the line:

*ASSEMBLE SOURCE -OPROG -P -L2 -B8

will assemble from file SOURCE, putting the object code into file PROG. The default listing level used will be 2, and the listing will be sent to the printer only. An 8 kilobyte memory buffer is to be used for the object code to speed up assembly.

The Assembler will leave the screen in whatever mode you have selected, so that for large assemblies you may need to change to mode 7 to provide room in memory for the symbol table. Shadow mode will automatically be selected if it is available, so that

on the B+, B+128 and Master series you will always have the maximum space available.

1.4.2 Alternative syntax forms for the command line

The example given above, and the output the Assembler will produce if you type

*HELP ASSEMBLER

detail the full form of the command line syntax. As you gain experience in using the Assembler, however, you might wish to take advantage of various ways of specifying the options in a more compact form.

Firstly, the Assembler allows you to specify the options all at once, without the need to put the "-" flag before each of them. Thus, you could type either

*ASSEMBLE SOURCE -M3 -W -L2

or the shorter, but less readable

*ASSEMBLE SOURCE -M3WL2

Secondly, the "-O" flag to specify an object file is optional. The Assembler will take the second parameter of the command as an object file name, unless it starts with a "-" character. Thus

*ASSEMBLE SOURCE -OPROG

and

*ASSEMBLE SOURCE PROG

are identical in effect. If you should ever need to start the object file with a "-" character (e.g. if you wish to use a temporary file system on a Master 128) you will have to specify the "-O" flag explicitly.

1.5 Assembly on a second processor

The 65C02 Assembler is compatible with both the external 6502 second processor, and with the Master Turbo-Card. Using a second processor will give a faster assembly due to the higher clock rate of the processor, and will also give you more memory for the symbol table.

A second processor allows you to organise your source files to take advantage of a source file memory buffer. Any source file that is less than 14 kilobytes in size is *LOADed into memory and assembled from

there: since the Assembler does not then need to wait for data to be fetched from disc, assembly will speed up dramatically. It is well worth arranging your program so each of its source files will fit into this buffer area.

The source buffer is automatically switched on when a second processor is in use.

1.6 The object code buffer

The Assembler allows you to select an object code buffer in memory. Code bytes generated from the source are written into this buffer, rather than being sent directly to the object file, and this gives a substantial increase in speed of assembly. Whenever the buffer is full, the assembly will pause in reading source to write the contents to the object file. It then resumes assembly until the buffer again fills, and so on.

You can specify the buffer size with the "-B" option. The flag is followed by a number between 1 and 16, giving the buffer size in kilobytes. Note that using a buffer means that there is correspondingly less space available for the symbol table.

The fastest assembly will be achieved when the object code fits entirely into the buffer. In this case, the Assembler will produce the object file with *SAVE at the very end of an assembly, which is substantially faster even than writing it in large blocks.

1.7 Returning to the calling language

When an assembly finishes, the Assembler will always restart the language ROM that was in use when you issued the *ASSEMBLE command. If

this language was BASIC or the Assembler's own CLI mode, there will be no difficulty: however, some languages such as WordWise or View immediately clear the screen when they start, so you may not have time to read the final lines that the Assembler displays on the screen.

To overcome this, the Assembler is able to pause when it ends until you press a key. It will do this whenever the previous language is not BASIC or the Assembler's CLI mode.

If you don't wish it to pause (you may for example not want to read the final screen, or the language may not clear the screen), simply specify a "-G" option in the command line. Now the Assembler will immediately restart the previous language without a pause.

There are two cases where the Assembler will always pause regardless of the "-G" option or the previous language. These are:

- When an error is detected in the *ASSEMBLE command you typed

  or

- On a machine with a shadow screen that was turned off when you began assembly. The Assembler will turn the shadow screen on automatically, and will always turn it off at the end: as this requires a mode change which will clear the screen, it will always pause.

1.8 Use of control keys during assembly

During an assembly you can use various control keys to check on progress or give commands to the Assembler.

- Function keys f0, f1, f2 and f3 will force the Assembler to change the source listing level to 0, 1, 2 or 3 respectively. Once you have pressed one of these keys the level you set is locked: the Assembler will process LST directives in the source, but will not implement them. This lets you, for example, change the list level during an assembly to check on what's happening, or to suppress a listing you decide you don't want. To cancel the locking of the list level and return to that set by the last LST directive processed, simply press CONTROL-L.

- CONTROL-H displays some help information to remind you about the use of control keys. The Assembler will pause after displaying the information: to resume simply press any key.

- CONTROL-N and CONTROL-O turn the BBC Computer's paged scrolling mode on and off respectively. You could use these keys, for example, to pause a listing on the screen after every screenful at some point during an assembly.

- CONTROL-P controls whether the Assembler sends output to a printer. If it is printing, pressing CONTROL-P will stop it: if it is not printing, CONTROL-P will start it sending output to the printer.

  If you have used the -F option to send output to a listing file, CONTROL-P can be used to switch output between the screen and the file, but will not affect any printer you may have connected to your machine.

- CONTROL-Q lets you find out where the Assembler is in the current file. It will tell you whether it is on pass 1 or 2, then give you the number of the line it is about to process. If the number is followed by "(M)", the source line is a macro call and the Assembler is currently expanding that macro.

- CONTROL-W reverses the current "wait after error" action set by the command line's "-W" option.

- SPACE pauses the assembly. You can use this to examine a listing on screen at your leisure. To resume assembly, simply press any

  key.

2. SOURCE PROGRAM FORMAT

The Assembler will read source from files produced by any editor or word-processor program, as described in section 1.3. The source is seen as a number of lines, each ending with a carriage return byte ($0D). Line feed characters are ignored, as are all other control characters and (unless you have used one of the configuration options detailed in Appendix 5) any with the most significant bit (bit 7) set. The general format of a line is:

label opcode operand comment

Depending on circumstances, the label and operand fields may be optional or mandatory; or they may need to be omitted. All lines may have a comment field, which is begun with a ";" or a "\" character.

If you include a label field, it must begin in the first character of the line. You may separate the fields from each other with any number of spaces and/or TAB characters. No line may exceed 132 bytes in length; any that do will be truncated.

Any line whose first non-space character is "*", ";", "!" or "\" is treated as comment. Lines beginning with "!" are treated specially if they occur within the definition of a macro, as explained in section 9.6.

2.1 Syntax of the line

2.1.1 The label field

A label consists of a string of characters, starting with a letter, and containing any combination of letters, numbers and the characters ".","$" and "_" (underscore). Any lower-case letters are translated to upper-case. Labels may be of any length, and all are significant unless you have specified the "-A" option in the command line, when only the first 6 are significant.

A label may be terminated with a ":" character that is not considered part of the label.

Examples of valid labels are:

```
OSWRCH
osbyte
Program.start
ITEM_33:
```

A label may be written in a line on its own: the value it is set to will be the value of the current address.

A variant of the label is the Sequence Symbol, described in section 10.1. This has the same format as a normal label, with the exception that the first character must be a "%" character.

Where the label is a macro name (i.e. in the MACRO directive) it may not exceed 8 characters in length.

2.1.2 The opcode field

The opcode field, if present, is separated from the label by one or more spaces or TABs. If there is no label, the field must be preceded by at least one space or TAB.

The opcode field can contain either a normal opcode mnemonic (such as LDA), an Assembler directive or pseudo-op (such as SFCOND) or the name of a macro that you have defined previously.

2.1.3 The operand field

For some opcodes and directives an operand field may be supplied. This will consist of one or more elements, separated by commas and optional spaces. Except within strings delimited by the single- or double-quote characters, spaces are not significant in the operand field. Thus you can use them to make complex numeric

expressions more readable: for example you could write

```
LDA   TABLE+(1+FRED/3*(BERT+$1E))
```

as

```
LDA   TABLE + (1 + FRED/3 * (BERT + $1E) )
```

if you wished.

2.1.4 The comment field

The comment field can be used to annotate the line. It is begun with a ";" or a "\" character: anything following is not processed by the Assembler.

2.2 The syntax of expressions

Throughout this guide you will see references to "expressions", for example in the definitions of directives. The syntax of these is as follows:

2.2.1 Numeric expressions

These are indicated in definitions as "<expr>", and consist of

terms separated by arithmetic operators. For example, a valid expression might be

```
FRED*(2+BERT)
```

The Assembler performs all calculations using 16-bit arithmetic. Any results or intermediate values that overflow will be truncated, and no warning will be given.

The various elements that you can use in numeric expressions are these:

a. Symbols

   These are the labels that you define in the label fields of lines, and in the evaluation of the expression they are replaced by the value of the label.

b. Decimal constants

   These are numbers, composed of the characters '0'..'9'.

   For example,

c. Hexadecimal constants

These are numbers, composed of hexadecimal digits '0'..'9' and 'A'..'F', preceded by a "$" or a "&" character.

For example,

$AF34

d. Binary constants

These are numbers composed of binary digits '0' and '1', preceded by a "%" character.

For example,

%10100111

e. Character constants

These are single characters, enclosed in single-quotes. The Assembler will use the ASCII code of the particular character.

You can specify that a character value is to be used with bit 7 set by preceding the character with a "^", and can specify a control character by preceding it with a "|". Thus the code for an "A" with bit 7 set is represented as '^A' and that for

CONTROL-A as '|A'.Since the "|" character actually inverts bit 6 of the character that follows, you can obtain the code for "DEL" (hexadecimal code 7F) by writing '|?'.

The character codes for "^" and "|" are obtained by writing the characters twice (i.e. as '^^' and '||').

The Assembler does not by default give you a means to specify that a character value is to be used as a control character and also with bit 7 set. You may find that this would be useful (if for example you wished to plant Teletext control codes such as $81 in a string), so the Assembler can be configured to permit this to be done. One of the options described in Appendix 5 lets you set up the Assembler so that the "~" character (which appears on a Teletext screen as a "divide" sign) causes the next character to be output as a control character with bit 7 set. Thus, to obtain the hex code $81, you would write '~A'. Naturally, once you have selected this option, you will need to double the "~" character as well as the other two special characters should you wish it to appear itself in a string.

Note that you must always include the final quote character.

For example,

```
'X'
```

f. Operators

A number of arithmetic operators can be used in expressions. They are divided into groups of varying priority, which, in decreasing order are:

```
() Parentheses

-  Unary minus
~  Unary 1's complement (NOT)

&  Bitwise AND
!  Bitwise OR

=  Equality
#  Inequality
>  Greater than
<  Less than

*  Multiplication
/  Integer division

+  Addition
-  Subtraction
```

The "=" and "#" operators return values of -1 and 0 for TRUE

and FALSE respectively.

Additionally, expressions may be prefixed with two unary operators > and <, which select the low-byte and high-byte of the result respectively. These operators have the lowest priority of all. Thus, for example, suppose you have defined a table starting at label TABLE. To load the address of the table into X (the low byte) and Y (the high byte), you would write

```
LDX   #>TABLE    ; load low byte
LDY   #<TABLE    ; load high byte
```

Note that the MOD and DIV operators used in the BBC Basic assembler to achieve this result cannot be used.

g. Current address

The current value of the address counter may be represented in

expressions by the "*" character.

For example:

```
*-3
```

2.2.2 Strings

These are indicated in definitions as <string>, and consist of one or more ASCII characters, enclosed in delimiters. Only the single- and double-quote characters (' and ") can be used as delimiters, and the start and end delimiter must be the same.

For example:

```
'This is a string'
"And so, you see, is this"
```

To include the ' or " character in a string, use the other as the delimiter.

Within a string you can obtain control characters and characters with B7 set in the way described above for character constants.

2.2.2 String expressions

These are indicated in definitions as <stringexpr>, and all involve comparisons between strings. The general format is

```
<string> <op> <string>
```

where <op> is one of the operators


```
=  Equality
#  Inequality
>  Greater than
<  Less than
```

3. THE OUTPUT LISTING

The assembly listing is produced in pass 2. You can control whether it appears or not, whether it goes to screen or printer, and the amount of detail it contains.

3.1 The source listing level

The main control you have over the listing is through the "source listing level". This is a value in the range 0..3 that you can set either with the LST directive, the -L command line option, or with function keys 0 to 3 during the course of an assembly.

The listing levels have the following meaning:

0 This level suppresses all listing except the reporting of errors.

1 This level will list all source lines that originate in source files, but not the expansions of macros.

2 This level lists all lines from source files and macro expansions, but not Macro Programming Language statements such as AIF.

3 This level lists all source lines.

By default, the Assembler sets the source list level to 0, so all you will see will be error reports. You can change the default setting in the *ASSEMBLE command by using the "-L" option: thus

```
*ASSEMBLE SOURCE -L2
```

would cause the Assembler to start with a default listing level of 2.

Within the source, you can control the list level with the LST directive. Thus:

```
LST  1
```

would set the list level to 1.

Simply writing

```
LST
```

with no operand field will reset the value to the default, without you needing to build this value into the actual source.

Additionally, you can override the list level in use while an assembly is in progress. Pressing one of function keys 0 to 3 will force the corresponding list level to be adopted, so that, for

example, you could force the Assembler to show you parts of an assembly to monitor what was being done. Pressing one of these keys will "lock" the list level to that selected, so that LST directives in the source will not be able to turn the listing off again. The directives are noted, though, and you can return listing level control to them at any time by pressing CONTROL-L.

3.2 Source listing format

The listing begins with the source lines, including any code bytes they have generated.

The first character on the line indicates where the line was read

from. If it came from an INCLUDE file or a CODE file the character will be "I"; if the line is from a macro it will be "M". Otherwise, the character is a space.

The next element is the line number, reset to 1 at the start of each source file.

Next comes the address field. This will normally contain the current value of the address counter; in some directives, though, such as LST, it contains the value of the operand expression.

Then comes a hexadecimal representation of the first 3 object bytes generated by the line; this is followed by the source line itself. By default, if the source line generated more than 3 object bytes, these will be listed on subsequent lines.

For example, a listing line might show:

```
I   23 AB34:C9 0A      CMP  #10
```

where the source line is the 23rd line in an INCLUDE file and reads "CMP #10". The code bytes generated are $C9 and $0A, and they are planted starting at address $AB34.

When listing a source line, the Assembler will expand any TAB characters it encounters. By default, TAB positions are defined at characters 8, 16, 25, 33, 41, 49, 57, 65, 73, 81, 89, 97, 115 and 123, with the first character of the source line being numbered 1. The default TAB positions have been chosen to give a neat listing layout using labels of up to 6 characters in length and typical operand field sizes. However, the first few TABs are not positioned at regular intervals, so you may find that you cannot easily lay out tables within comments.

You can overcome any problems, and choose the layout most suitable for your source, with the TABS directive. This lets you redefine the TAB positions used: for example

```
TABS  9,17,25,33,41,49,57
```


would set TAB stops at regular positions, with each field being 8 characters wide. You can return to the default TAB stops at any time with the directive

```
TABS
```

with no operands.

The Assembler can be customised to change the default TAB positions to any value you require: this is described in Appendix 5.

3.3 Listing object code bytes

As mentioned above, the first three code bytes generated by a source line are listed on that line, and any others will appear on subsequent lines. This may produce a lot of output, particularly if you define lots of strings, so the Assembler gives you control on how much it lists.

The amount is controlled by the "code listing level". This is a value in the range 0..2 that you can set either with the CLST directive or the -C command line option.

The code listing levels have the following meaning:

0 Only the first 3 bytes generated by a line are listed.

1 All bytes are listed from all lines except the CODE directive.

2 All bytes are listed, however they are generated.

By default, the Assembler sets the code list level to 1. You can change the default setting in the *ASSEMBLE command by using the "-C" option: thus

```
*ASSEMBLE SOURCE -C0
```

would cause the Assembler to start with a default code listing level of 0.

Within the source, you can control the code listing level with the CLST directive. Thus:

```
CLST  2
```

would set the list level to 2.

Simply writing

```
CLST
```


with no operand field will reset the value to the default, without you needing to build this value into the actual source.

3.4 Symbol table listing format

If the source listing level is not zero at the end of the assembly, the Assembler will list the symbol table, showing you the value of all the symbols defined in the source.

The symbols are listed in alphabetical order, and only the first 13 characters of each will be shown. The value of each symbol is shown in hexadecimal.

If the value is shown as "????", the symbol was used but was not defined. If the value is "****", the symbol was defined more than once. A "-" character after the value indicates that the symbol was defined, but was not used anywhere else in the source.

3.5 Error reports

The Assembler will list all lines that generate errors, regardless of the listing level. The errors are reported with self-explanatory text messages following the line.

Where appropriate, the Assembler will attempt to indicate where on the line the error occurred. The text you might see would look something like:

```
****** Err : Undeclared symbol at about character 19
```

This indicates that the erroneous symbol is at about the 19th character in the line, which starts at character 1. Remember, though, that the number refers to the source line: the listing line will have expanded any TAB characters you used, and the number will then not correspond to what you see on the page.

3.6 Output page control

The Assembler provides a number of directives that you can use to specify the exact format of the output page, and to let you tailor the listing to your exact needs.

3.6.1 Setting page dimensions

The PAGE directive defines the size of the page you are using. It takes two parameters, which are the total depth of the page in lines, and the width in characters you wish printed.

Thus,

```
PAGE   66,132
```

tells the Assembler that the page is 66 lines deep, and that you wish lines to contain a maximum of 132 characters. The Assembler will always allow a small gap at the end of the page to avoid the

perforations, and will truncate any lines longer than the maximum width you specify.

The value of the width you define with PAGE becomes effective at once. The depth, though, is not used until the next page throw, so you would normally follow the directive with a TTL or a SKP H directive.

By default, the Assembler will print lines of 80 characters on paper that is 66 lines deep. It moves to the top of a new page by sending a Form Feed ($0C) character to the printer.

3.6.2 Setting the page title

The TTL directive lets you set up a page title that is printed on the top of every listing page. The title can be up to 20 characters long, and can contain any printable text.

For example,

```
TTL  'Screen Dumper'
```

will print "Screen Dumper" at the top of each page.

The TTL directive will cause a page throw to occur immediately, and the next page output will use the title it defines.

3.6.3 Setting the timestamp

If you are using a Master 128, the Assembler will automatically print the current date and time at the top of each output page, taking the values from the built-in real-time clock. The other BBC models do not contain clocks, so the TIME directive lets you simulate the effect by defining the string used.

The TIME directive allows you to define a string of up to 25 characters that will appear in the page header. The string can contain any printable characters: it is not constrained to be a date and time value, so you can use it as a sub-title if you wish.

For example, you could use


```
TIME  'Friday at 1030'
```

or

```
TIME  'First Module'
```

The directive is ignored on a Master 128, so there is no need to change a source that is being assembled on both a model B and a Master 128.

It is possible to customise the Assembler to use a real-time clock fitted to a machine other than a Master 128: for details see Appendix 5.

3.6.4 Page layout directives

There are a number of directives that let you lay the output page out in more detail, so you can delimit sections of source for easier inspection and configure the axact appearance of each line.

The REP directive provides a convenient way of separating sections of code in the listing. For example,

```
REP  80
```

will print a line of "*" characters in the listing for you (and takes much less space in the source file than that line itself would).

The CHR directive lets you change the character used to make the line, so that, for example you could select lines of "-" characters with

```
CHR  -
```

The SKP directive lets you break the listing up for easier reading. For example,

```
SKP  5
```

would leave a 5 line space on the page. The format

```
SKP  H
```

causes the Assembler to start a new page. Note that the SKP directive itself will never be listed unless it contains an error.

3.7 Controlling the output destination

By default, the Assembler sends the assembly listing to the screen, in whatever mode you have selected. You can, though, choose to direct the assembly listing to a printer; or to output it to a file.

3.7.1 Sending the listing to a printer

You can direct output to the currently-selected printer by specifying the "-P" option in the command line. Thus, for example,

```
*ASSEMBLE SOURCE -L2 -P
```

would assemble from source file SOURCE and send the listing, at level 2, to the printer only.

At any time during an assembly, you can change the output destination by pressing CONTROL-P. This will switch the output destination between printer and screen: you can do this as often as you wish.

3.7.2 Sending the listing to a file

You can direct the assembly listing to a file by using the "-F" option in the command line. Thus, for example,

```
*ASSEMBLE SOURCE -L1 -F:3.LISTING
```

will produce an assembly listing at level 1, and will send it to the file :3.LISTING only.

Output sent to a file is laid out exactly as if it were going to a real printer. It will be broken up into pages, with the page depth and width controlled by the PAGE directive. After assembly, you can for example print the file with

```
<CTRL-B>
*TYPE :3.LISTING
<CTRL-C>
```

Any printer control codes that you may send with the SYSVDU directive are not stored in the file, so you can output the file to any printer without the need to edit it out unsuitable codes first.

If you have used the "-F" option, the action of CONTROL-P during assembly changes slightly. You can now use it to switch output

between the screen and the file, allowing you, for example, to suppress listing of some parts of a source. It has no effect on any printer that you may have fitted.

If you use both the "-P" and "-F" options on the command line, only "-F" has effect.

3.7.3 Printer control

You will often need to send a sequence of control bytes to your printer before starting an assembly (for example to set it into condensed print mode). The assembler gives you three directives to accomplish this automatically: SYSVDU sends a control sequence on both passes, and SYSVDU1 and SYSVDU2 do the same, but only on pass 1 or pass 2 respectively. If you have not used the "-P" command line option to produce a printed listing, the directives will not send any output to it: there is no need to alter your source to suit the times when you are printing or not printing.

To output a control sequence, simply write the bytes as operands to one of the directives, exactly as you would do in BBC Basic's VDU command. Thus

```
SYSVDU1  1,27,1,'E'
```

sends the character ESCAPE (whose ASCII code is 27 decimal) followed by 'E' to the printer only, on pass 1 of the assembly.

Note that you do not need first to send a code of "2" to activate printer output: the assembler itself ensures the bytes go to the printer if one is in use. The "1" codes in the example above act exactly as in BASIC's VDU command, and send the following character to the printer only. You can if you wish use SYSVDU to write to both printer and screen: thus

```
SYSVDU   'A','B'
```

sends characters 'A' and 'B' to both. However, the DISP family of directives give you a much better means of outputting messages to the screen.

The directives do not send any bytes to a listing file that you may have selected with the "-F" command line option. This means that you can safely print a listing file on any type of printer without the need to edit out possibly inappropriate control sequences first (but, of course, you may need to set the printer up yourself before printing the listing).

4. DIRECTIVES DEFINING DATA AND CONSTANTS

The Assembler contains a number of directives that allow you to define values in the object file. You can specify byte values, two-byte or word values, and character strings, and also set up symbols to hold constants.

4.1 Defining constants

Assigning symbols to various constant values is a considerable help in writing assembly-level programs, both in terms of making a source more readable and also in helping when you need to change it.

For example, you could refer to an Operating System routine by writing its address $FFEE every time, or you could declare a symbol OSWRCH to stand for that name.

You can declare symbols to use as constants with the EQU directive. In the example above, you would write

```
OSWRCH  EQU   $FFEE
```

and then call the routine as, say

```
JSR   OSWRCH
```

You can write any numeric expression in the operand field of the EQU directive, but the expression must not contain any forward references.

4.2 Byte and word directives

There are 6 directives that plant byte and word values. DFB plants bytes, and DFW and DFDB plant word values. For convenience, and compatibility with other assemblers, the directives can also be written as DB, DW and DDB respectively.

All the directives can be given one or more expressions in the operand field, separated by commas, and these expressions give the values of the bytes or words to be planted. If you specify a label in the label field of the line, it is set to the address of the first byte of the first value planted.

4.2.1 Byte directives

The directive DFB (or its equivalent DB) allows you to define single-byte values which are written directly into the object file. The operand field is one or more expressions, each of which

should lie in the range 0..255 .

For example,

```
DFB  1,2,$FF,FRED+1
```

will write 4 bytes to the object file: the values will be 1, 2, 255 and the value of FRED+1.

4.2.2 Word directives

The directives DFW (or its equivalent DW) and DFDB (or its equivalent DDB) allow you to define two-byte or word values which are written directly into the object file. The operand field of each is a list of one or more expressions, each of which should lie in the range 0..65535.

The difference between the directives lies in the order in which the two bytes are written to the object file. The DFW directive writes the bytes in low-byte, high-byte order; the DFDB directive writes them in high-byte, low byte order.

Thus, for example,

```
DFW   $1234,$ABCD
```

will output the bytes $34, $12, $CD, $AB to the object file in that order, but

```
DFDB  $1234,$ABCD
```

will output the bytes $12, $34, $AB, $CD.

4.2.3 Using repeat counts

Occasionally, you will need to use the byte or word directives to plant a number of bytes or words containing the same value. For example, you might need to write

```
DFB  5,5,5,5,5,5,5
```

to plant 7 bytes containing the value 5.

For small numbers of repeats, this is no problem, but it can be onerous if, say, you needed 49 bytes containing 5.

The Assembler gives you a very convenient shorthand way of repeating values in all the byte and word directives. Every expression you use can be prefixed with a "repeat count", which tells the Assembler to plant the value itself more than once. The

repeat count is specified in "[]" brackets: thus, to plant the 49 bytes mentioned above, you could simply write

```
DFB   [49]5
```

Like the value itself, the repeat count can be any expression, but it must not contain forward references. Thus you could write

```
DFB  [(COUNT+27)/2]$FF
```

You can freely mix values with and without repeat counts in any of the byte and word directives.

A repeat count value of zero will cause no bytes at all to be written to the object file.

4.3 Character string directives

The four directives ASC, CASC, STR and CSTR allow you to plant character string values into the object file.

All the directives require you to specify a character string, enclosed in single- or double-quotes, in the operand field. If you include a label in the label field of the line, it is given the value of the address of the first byte planted.

4.3.1 The ASC and STR directives

These directives plant simple strings in the object file. ASC plants the string exactly as you specify it; STR plants the string, and then automatically plants a carriage-return byte (code $0D) immediately after it.

Thus

```
ASC  'ABC'
```

plants the bytes $41, $42, $43 in the object file, and

```
STR  'ABC'
```

plants the bytes $41, $42, $43, $0D.

4.3.2 The CASC and CSTR directives

These directives plant "counted strings" in the operand field. They are similar to ASC and STR, except that the bytes planted are preceded by a single byte giving the length of the string.

Thus, to use the same examples as above,

```
CASC  'ABC'
```

plants the bytes $03, $41, $42, $43 in the object file, and

```
CSTR  'ABC'
```

plants the bytes $04, $41, $42, $43, $0D.

Note that the count byte planted by CSTR includes the $0D byte that the directive adds itself.

4.3.3 Planting special characters

The strings you give as the operands to ASC, CASC, STR and CSTR must of course be made up of printable characters in the source file, but the Assembler allows you a way of specifying control characters and characters with the most-significant-bit (bit 7) set to 1.

To specify a control character, you should precede the character itself with a "|" character. Thus, if you wish to plant the value of CONTROL-A, the string should contain "|A". This method is identical to the way in which you specify control characters in Operating System commands such as *KEY.

To specify a character with bit 7 (the most significant bit) set, precede it with a "^" character: thus, to plant an "A" with bit 7 set, specify it in the string as "^A".

For example,

```
ASC  'A|AB^B'
```

plants the bytes $41, $01, $42, $C2 in the object file.

Should you need to specify the "|" or "^" characters, you should double them in the string (i.e. write them as "||" and "^^")

The Assembler does not by default provide a means to plant characters as control characters also with bit 7 set. However, you can use a configuration option described in Appendix 5 to let you do this: selecting the option makes the "~" character plant the character following as a control character with bit 7 set. Thus

```
ASC  "~A"
```

would plant the byte $81 (i.e. a CONTROL-A with bit 7 set).

4.4 The HEX directive

This directive is rather a hybrid of the DFB and ASC directives. It plants a series of byte values into the object file, but you specify the values as a string of hexadecimal digits.

For example,

```
HEX  '1234ABCD'
```

plants the byte values $12, $34, $AB, $CD into the object file.

The string you supply must contain valid hexadecimal digits (i.e. '0'..'9' and 'A'..'F'). You can give the letters in either upper- or lower-case. If the string contains an odd number of characters, the Assembler will automatically add a '0' to make the final hexadecimal value complete.

4.5 The CODE directive

The code directive gives you a convenient way of including things like screen dumps or already-compiled relocatable subroutines into your programs.

Suppose, for example, you had prepared a mode 7 screen dump with some Teletext editor system, and wanted to use this as a banner page when your program started. One way of including this into the code would be to examine the dump and convert it into the appropriate DFB directives, planting each byte explicitly by hand. However, this might be a very time-consuming process, and would be very error-prone.

Using CODE, though, removes the need to translate the dump into DFB or similar directives. If, for example, the dump was in file BANNER, writing

```
CODE  BANNER
```

would cause the Assembler to read it in and copy the bytes directly to the object file, without processing them in any way.

By default, the listing will show only the first 3 bytes included from the CODE file, since otherwise a considerable amount of paper might be used. If you do want to see all the bytes in the listing, use the CLST directive to set the code listing level to 2 before using the CODE directive itself.

You cannot use the CODE directive within a macro, or from an

INCLUDE file.

5. OBJECT FILE CONTROL DIRECTIVES

The Assembler contains a number of directives that control the object file. They let you define the actual address that the code generated will run at, map symbol definitions into workspace, and set the load and execution addresses for the final result.

5.1 Defining the current address

In almost all programs you assemble, you will need to tell the Assembler the actual address at which the resulting code is to run. It will then be able to assign the correct address values to labels.

You define the current address with the ORG directive. For example, you could write

```
ORG    $1900
```

before the first code line in the source. This would set the address value to $1900, and the Assembler would produce code based on that address.

You can specify the address in the operand field of the ORG directive as any expression you like: however, since the Assembler must know the value to use on pass 1, the expression cannot contain forward references.

Note that the ORG directive permits you to specify only a two-byte or 16 bit address value. Disc filing systems use a 32 bit address value to control which processor a file is to be loaded into, and to set the top two bytes with the Assembler you should use the LOAD and EXEC directives, described in section 5.4.

You can use as many ORG directives as you wish within your program source, and you can move the current address value to anywhere you wish. However, you should remember that the ORG directive has no effect on where in the object file bytes are written. Thus, the source lines

```
ORG   $1900
LDA   #1
ORG   $3000
LDA   #2
```

would set the address to $1900, and write the two bytes for the LDA

instruction to the object file. It then sets the address to $3000 and writes the bytes for the second LDA instruction: however, the four bytes will be contiguous in the object file - the Assembler will not generate a gap. You could use this effect, for instance, to write an object file containing many sections of overlay code - each section

would be assembled to run at the same address, but they would lie one after the other in the object file.

If you do require to leave a gap in the object file, you can use the DS directive to generate the required gap yourself (see section 5.3.1). There is no way of making the Assembler move backwards in the object file.

5.2 Reading the current address

You can use the value of the current address counter within expressions by referring to it with the special symbol "*". This can be used exactly as any other symbol in expressions, so that, say, you could work out the size of a data table with

```
TABLE   DFB   1
        DFB   2
        DFB   3
T.SIZE  EQU   *-TABLE
```

where T.SIZE will be set to the number of bytes defined in the preceding table.

5.3 Using dummy sections

A "dummy section" is a convenient way of laying out symbols within the workspace that your program needs. For example, suppose your program needed some page zero workspace. You could build the actual numeric addresses into the source, but it's much better programming practice to define some symbols to identify the locations.

One way of doing this would be to use the EQU directive, which simply sets up a symbol and gives it a value. Thus you could write

```
WORK0  EQU   $00
WORK1  EQU   $01
WORK2  EQU   $02
```

to set up three symbols. This is a perfectly adequate technique, but suffers from the disadvantage that you would have to change a lot of the source if, say, you ever wanted the page zero workspace to start at some place other than address $00.

A better technique is to define workspace in a dummy section, which is a region of the source between DSECT and DEND directives. DSECT instructs the Assembler that a layout is being defined: it will process anything that follows it, but it won't generate any code for the object file. All it will do is work out what addresses any labels will take.

Thus, you could write the above example as

```
        DSECT
WORK0   DFB   0
WORK1   DFB   0
WORK2   DFB   0
        DEND
```

Here each of the workspace locations has been defined as one byte long with a DFB directive. Since the Assembler generates no code within a DSECT, the actual value you place in the operand field is quite arbitrary. This technique also lets you see clearly that you've set the locations up as one-byte values. If you wished, say, to make them two-byte values, you could change the DFB directives to DW. The Assembler would then allow two bytes to each label, and the source would show this clearly.

In the example above, we have implicitly taken advantage of how the Assembler handles the address value within a DSECT..DEND block. When it meets the first DSECT in a source, the Assembler makes a note of the address value it is using in the code, and resets it to zero for the DSECT..DEND block. When it meets the DEND, it resumes using the address it saved at the start.

On the second occurrence of a DSECT, the effect is the same, except that now the address within the DSECT..DEND block resumes at the value it reached in the last block.

If you don't wish to operate in this way, you can set the address value with the DSECT..DEND block yourself with the ORG directive. Thus, if you wanted your page zero workspace to begin at some other location, you could write the source as

```
        DSECT
        ORG   $70
WORK0   DFB   0
WORK1   DFB   0
WORK2   DFB   0
        DEND
```

5.3.1 The DS directive

Quite often you will need to define a large area of workspace within a DSECT..DEND block, for example as a source line input buffer. The DFB and DW directives are convenient ways of laying out byte and word values, but there are no built-in directives to lay out, say, a 256-byte buffer.

To do this, you can use the DS directive, which lets you lay out any number of bytes. For example, to lay out a 256-byte buffer you could write


```
        DSECT
        ORG    $A00
BUFFER  DS     256
        DEND
```

The operand field of the DS directive tells the Assembler how much space you wish to lay out.

Normally the DS directive is used in DSECT..DEND blocks, but you can also use it in the code area of a program as well. In this case, the Assembler will write the appropriate number of bytes containing the value zero to the object file.

5.4 Defining object load and execution addresses

Every machine-code program on a BBC Computer needs to have a load address and an execution address associated with it. These values are stored in the file's catalogue entry: when you then *RUN the file, the filing system will load it to memory starting at the load address, and will enter it with a JSR instruction going to the execution address.

The addresses also tell the filing system whether the program is to be run on a second processor, or on the BBC Computer itself. The filing system regards load and execution addresses as 4 byte (32 bit) values. The lower two bytes or word of each address represent an actual address in memory and the top two bytes or word of the address is a value that indicates which processor the file is to be loaded on. The Assembler can handle only two-byte values in its arithmetic, so it is necessary to set the two parts of each address separately.

5.4.1 Specifying the low word of the addresses

There are two ways of setting up the low words of the addresses when you assemble programs. By default, the Assembler will set the low words of both the load and the execution addresses to be the value set by the first ORG directive that is not in a dummy section, and in many cases this is all you need do.

If, though, you require something else, you can use the LOAD and EXEC directives to set the low word of the load and execution addresses respectively. Thus, for example,

```
LOAD  $1900
EXEC  START
```

would set the low word of the load address to be $1900, and the low word of the execution address to be whatever address the

label START is defined as.

5.4.2 Specifying the processor

If the top word of the load address is $FFFF, the file is loaded into the BBC Computer's own memory, whether or not you are using a second processor. If the top word is 0, though, the file will be loaded into the second processor if you are using one.

By default, the Assembler produces object files with the top word of the addresses set to $FFFF, so the files will load to the BBC Computer's memory. The MSW directive, though, lets you change this value: thus

```
MSW   0
```

lets you change the address to 0. You should always include this directive if you write programs to run on a second processor.

6. SOURCE FILE CONTROL DIRECTIVES

The Assembler provides a number of directives that allow you to control the source files which you are assembling. This lets you split your source up into more than one file.

6.1 Chaining source files

If you are writing a large program, you will find it convenient to split the source into a number of small files, rather than keeping it in one large one. This approach not only makes editing easier, but also helps you structure the source to reflect the organisation of the program. Additionally, if you have a second processor, keeping source files to less than 14 kilobytes in size will give a very considerable increase in assembly speed (see section 1.5).

You can instruct the Assembler to assemble multiple files with the CHN directive. This tells it to close the current source file and open the one specified in the operand field.

Thus, you could write

```
CHN  FILE2
```

as the last line in FILE1. The Assembler will close FILE1, and continue assembly with FILE2. Note that if the CHN directive is not the last line in a file, anything following it will be ignored.

You can use CHN as many times as you wish in an assembly.

6.2 Including one file into another

A different way of using multiple source files is to use the INCLUDE directive. This tells the Assembler to start assembling

another source file; unlike CHN, however, the first file is not closed. When the Assembler comes to the end of the included file, it resumes in the original file at the line after the INCLUDE directive.

One use of this directive might be to include a file of standard definitions or routines into a source. You might then write

```
INCLUDE  STDSUBS
```

to assemble the source of some standard routines.

Lines that result from assembling an included file are marked in the listing by having an 'I' in the first column.

Note that you cannot use an INCLUDE directive inside a file that

you are including already. Nor can you use the CHN and CODE directives.

7. CONDITIONAL ASSEMBLY

The 65C02 Assembler has full conditional assembly facilities, allowing you easily to change the source being compiled.

7.1 Assembly conditional on expressions

Conditional assembly that depends on the value of an expression is achieved by using the IF, ELSE and FI directives. The most general form of construction is this:

```
IF   <expr>
   .
   TRUE block
   .
ELSE
   .
   FALSE block
   .
FI
```

<expr> can be any numeric expression, as described in section 2.2, but it must not contain any forward references. If <expr> is non-zero, the Assembler takes the condition as TRUE, and will assemble the lines in the TRUE block. On coming to the ELSE it will ignore all the subsequent lines, until it reaches the FI; here assembly will continue normally once more.

If <expr> is zero, the condition is FALSE. The Assembler will

ignore the lines that follow until it reaches the ELSE; it then will assemble the lines in the FALSE block.

If you wish, you can omit the ELSE and the FALSE block of code, to form

```
IF   <expr>
   .
   TRUE block
   .
FI
```

Here, if the condition is FALSE, the Assembler will ignore all lines up to the FI, so no code is assembled in this case.

For example, you might have a symbol DEBUG.MODE defined at the very start of the source to show if debug code is to be assembled or not. This might be defined as

```
DEBUG.MODE  EQU   -1    ;use -1 for TRUE, 0 for FALSE
```

Then, throughout the rest of the source, you can put the debug code in an IF condition of the form

```
IF   DEBUG.MODE
   .
   debug mode version
   .
ELSE
   .
   non-debug mode version
   .
FI
```

Depending on circumstances, you may wish to reverse the condition tested to include some code if debug mode is not selected. You can use the unary 1's complement or NOT operator here: thus

```
IF   ~DEBUG.MODE     ;read as "IF NOT DEBUG.MODE"
   .
   non-debug mode version
   .
FI
```

includes code if DEBUG.MODE is set to 0.

The IF..ELSE..FI construction can also be written using the directives DO..ELSE..FIN (or with any combination of them)

7.2 Assembly conditional on the existence of symbols

Two variants of the IF directive, IFDEF and IFNDEF, allow you to test for the existence or otherwise of symbols. IFDEF is TRUE if the symbol in the operand field does exist; IFNDEF is TRUE if it does not.

Thus the example above might have been written as


```
IFDEF  DEBUG.MODE
   .
   debug mode version
   .
ELSE
   .
   non-debug mode version
   .
FI
```

where the mere existence of the symbol DEBUG.MODE would select the debug code, regardless of its value.

7.3 Nesting conditionals

The Assembler allows you to "nest" conditionals. Within a nested condition, an ELSE directive will be associated with the immediately preceding IF, IFDEF or IFNDEF.

7.4 Listing conditionals

By default, the Assembler will list all the lines that it skips in a conditional, printing an "S" in the address field.

To save paper, or make the listing clearer, you can suppress listing of skipped lines with the SFCOND directive, or by specifying the "-S" option in the command line. You can re-instate the listing of false conditional branches at any time with the LFCOND directive.

8. PROGRESS REPORTING

The Assembler supports several directives that allow you to output progress reports to the screen and to a printer or a listing file to show the course of an assembly.

8.1 The DISP, DISP1 and DISP2 directives

These all display a message on the screen, and also send it to a printer or to a listing file. DISP displays the text on both passes; DISP1 and DISP2 display the text on pass 1 only and pass 2 only, respectively. You can use INFO as a synonym of DISP2 is you wish.

The use of the directives is the same: for example, you might write

```
DISP  'Assembling debug code'
```

to display

```
---- Assembling debug code
```

on the screen.

The message you display can contain printable text, and you can also include some control characters. You specify them in the same way as you would in any other string, by preceding them with a "|" character. The control characters you can use are:

```
|M and |J  - start a new line
|G         - make a beep sound
```

For example

```
DISP   'Line 1|MLine 2|G'
```

displays a 2 line message and rings the bell.

Additionally, the text can contain numeric expressions that are evaluated as the message is output. For example, this would let you output a message to report the size of a section of code.

Within the text, you can specify an expression in one of two ways.

```
%D(<expr>)
```

will evaluate <expr> and display the result in decimal, and


```
%X(<expr>)
```

will display it in hexadecimal.

You can write any numeric expression you wish within the parentheses, but it must not contain forward references.

For example, to report the size of a section of code between labels START.CODE and END.CODE, use:

```
DISP2  'Size of code is %X(END.CODE-START.CODE) bytes'
```

8.2 The WAIT, WAIT1 and WAIT2 directives

These output messages in the same way as DISP, DISP1 and DISP2, but then suspend the assembly until you press a key on the keyboard.

You could thus use these directives to enable you to change source discs during an assembly.

8.3 The QUERY directive

This offers a convenient way of setting up conditions for conditional assembly.

QUERY operates only on pass 1 of the assembly. It displays a message on the screen in the same way as the directives above, then waits for you to type in a line. The line is treated as a numeric expression and evaluated, the result being set as the value of the symbol specified in the line's label field. If the expression is invalid, or contains a forward reference, the question will be repeated.

For example, suppose that your source contains optional debug code, and you select this with IF conditions of the form

```
IF    DEBUG.MODE
   .
   debug code
   .
FI
```

The value of DEBUG.MODE would normally be -1 to select debug code, or 0 if it were not needed.

Rather than define DEBUG.MODE in the source, and thus have to edit the source every time you wished to change it, you can make the Assembler ask for the value at the start of an assembly. A line such as:

```
DEBUG.MODE  QUERY  'Use debug mode'
```

will output the question

```
---- Use debug mode?
```

on pass 1. You can now type in

```
-1
```

to select debug code, or

```
0
```

to suppress it.

Since the line you type is an expression, and can contain symbols, you can make this more friendly by changing the source to include

```
YES   EQU   -1
NO    EQU   0
```

before the QUERY directive. Now you can reply either "YES" or "NO" to the question: the Assembler will evaluate the line and take the value of the symbol you type.

8.4 The STOP directive

STOP provides a useful way of abandoning an assembly part-way through. It operates on pass 1, and like DISP1, it displays a text message. Afterwards, though, it aborts the assembly with the error message "Stopped"

9. WRITING SIMPLE MACROS

The Assembler contains a powerful macro facility, allowing you to write very complex and sophisticated macros. In this chapter, we shall examine the simpler aspects of macros.

A macro is a way of producing a number of Assembler source lines merely by specifying its name in the opcode field of a line. For example, your source might contain many occurrences of the lines:

```
LDA   VAR
CLC
ADC   #2
STA   VAR
```

You could shorten the source, and make it easier to follow, by replacing each occurrence with a macro call: you might set up a suitable macro with the statements

```
ADDTWO  MACRO
        LDA   VAR
        CLC
```

```
        ADC   #2
        STA   VAR
        ENDM
```

Then, instead of writing the four lines of code, simply write one line

```
        ADDTWO
```

The Assembler will expand the macro definition, and will automatically generate the four lines for you.

9.1 Using macro parameters

Although this is useful in itself, the macro as shown above is rather limited. It couldn't, for instance, be used to add 2 to the variable COUNT instead of VAR.

In order to provide this sort of flexibility, you can pass information to macros as "parameters". You could re-write the macro above to read


```
ADDTWO  MACRO
        LDA   @P1
        CLC
        ADC   #2
        STA   @P1
        ENDM
```

Here we have replaced the occurrences of VAR by the string "@P1". Now, when the Assembler expands the macro, it will replace all occurrences of "@P1" with parameter number 1 of the macro call. To add 2 to VAR, you would then write

```
        ADDTWO  VAR
```

and to add 2 to COUNT, you would write

```
        ADDTWO  COUNT
```

You can supply up to 9 parameters when you call a macro, and they are indicated in the body of the macro by @P1, @P2, @P3 and so on.

(Note that you can also specify the parameters as @1, @2, @3 and so on, omitting the "P". This is adequate for existing code; however, for new programs you should use the "@P1" form, as this is necessary when you come to use the Macro Programming Language)

9.2 Specifying macro parameters

You can specify anything you want as parameters to macros. A macro can have up to 9 parameters, separated by a comma and optional spaces.

A macro with 3 parameters could be called with lines like

```
CHECK  1,FRED,27
CHECK  1 ,  FRED  , 27
```

and so on.

Normally, the Assembler will remove leading and trailing spaces from each parameter. If you require leading or trailing spaces, or if the parameter has to include a comma, you will need to specify it as a string, delimited by single- or double-quote characters. Thus, a macro call might look like

```
THING  'Here, is a comma'
```

and "@P1" will be replaced in the macro body with the characters

```
Here, is a comma
```

Note that the string delimiters are not taken as part of the parameter proper.

9.3 Nesting macros

You can call macros from within macros, up to a depth of 5. If you attempt to nest deeper than that the Assembler will flag an error.

9.4 Redefining opcodes and directives

The Assembler allows you to set up macros to redefine any opcode or directive. For example, you might want to redefine the JSR (Jump-to-Subroutine) opcode to automatically save the registers before entering the subroutine. You could do this by declaring a macro called JSR thus:

```
JSR   MACRO
```

```
        PHA
        TXA
        PHA
        TYA
        PHA
        JSR   @P1
        ENDM
```

Now, whenever the Assembler comes across a line with JSR in the opcode field, it will expand the macro JSR rather than obeying the opcode. It will plant the code to save the registers, and then will come to the line

```
        JSR  @P1
```

in the macro.

Here, because it is already in a macro, the Assembler will not use the macro JSR. Instead it will assemble the opcode JSR, planting the code to enter the subroutine.

9.5 Labels within macros

Suppose you wish to write a macro that includes a branch of some sort. You might write the macro definition as:


```
THING  MACRO
       LDA    @P1
       BEQ    ZERO
       EOR    #$FF
ZERO   STA    @P2
       ENDM
```

The first time the macro is called, it plants the code bytes, and defines the label ZERO as the destination of the BEQ instruction. On a subsequent call, though, the macro will produce the same code, and will attempt to define the value of ZERO again. This of course will fail, since it already exists from the first macro call.

The Assembler provides a way round this problem, by giving you a way of generating unique labels. Every time a macro is called, the Assembler sets up what you can regard as a special parameter on your behalf, which contains a string that is different for every macro

call. This string is substituted, in the same way as ordinary parameters, by writing "@$MC" in the line.

Thus, the above macro could be changed to be:

```
THING     MACRO
          LDA    @P1
          BEQ    ZERO@$MC
          EOR    #$FF
ZERO@$MC  STA    @P1
          ENDM
```

Then, on the first macro call, every occurrence of ZERO@$MC might be changed to ZERO1X1. On the next call, they become ZERO2X1, so that there is no clash between the macros.

10. THE MACRO PROGRAMMING LANGUAGE

A very powerful feature of the Assembler is its Macro Programming Language. This allows you considerable control in how macros are expanded - you can construct loops, manipulate macro parameters and perform several other functions that allow you to build macros of great power.

Although this facility is mostly intended for use within macros, many of its facilities can also be used outside macros to great effect, as this section will explain.

The Macro Programming Language's facilities build on two source

language features known as Assembly Time Variables and Sequence Symbols.

10.1 Sequence Symbols

These are "place markers" within your source files or within macros that the Macro Programming Language uses in loops. Using directives such as AGO and AIF, you can make the Assembler move up or down within a file or a macro, letting you repeatedly assemble some parts of the source or totally omit others.

Sequence Symbols are very similar to the labels that are part of the source proper, and they can contain the same characters. To distinguish them, Sequence Symbols always begin within a "%" sign in the first character of the line. The Sequence Symbol should be the only thing on the line: if you do put anything else there the Assembler will ignore it.

To take an example of how Sequence Symbols could be used, suppose your source file contained the lines

```
      AGO    %SKIP
      ASC    'These lines will never '
      ASC    'get assembled'
%SKIP
      ASC    'But this one will'
```

The Assembler will encounter the AGO directive, and will then ignore everything in the source file until it finds the Sequence Symbol %SKIP. It will then resume its normal processing.

Although this example will actually work, the technique isn't greatly useful, as ignoring source lines can be done just as easily with the IF..ELSE..FI construction. However, AGO (and the various conditional skips such as AIF) also allows you to go backwards in the

source or macro - there is no other way of achieving this.

The Sequence Symbols used in any file or macro are quite independent of those in any other; thus you can safely use ones of the same name in every file and macro, if you wish.

10.2 Assembly Time Variables

Assembly Time Variables, or ATVs, are string variables that the Assembler itself uses while assembling your source files. As it reads the program source (either from a file or from the definition of a

macro) the Assembler continually looks for references to ATVs. These are replaced - before the line is assembled - with the contents of the ATV, thus allowing you to vary the source that is actually processed.

You can use ATVs in many ways. For example, the first line of a source might set an ATV string to hold the version number of the program you are assembling; the Assembler will then automatically replace every reference to that name with the string. Some ATVs are created by the Assembler itself, and let you incorporate such things as the name of the source file into the source itself.

The main use, though, is in controlling loops within macros and within source files.

10.2.1 Creating Assembly Time Variables

ATVs have names similar to the variables that form part of the source proper, and you can manipulate them with various directives.

10.2.1.1 Local and Global ATVs

There are two types of ATV: local and global ones.

a. Global ATVs exist for the whole of the assembly, and can be used anywhere. They are created and manipulated with the ASTR, ASET, ASTRIP and ALEN directives, which you can use even inside macros - the ATVs they create will continue to exist after the macro finishes.

b. Local ATVs are created and manipulated by the MSTR, MSET, MSTRIP and MLEN directives. These directives can only be used inside macros, and the ATVs thy create can be used only within the same macro - they cease to exist at the end of the macro expansion. You would use these, for example, in controlling loops within a macro.

   In fact, you have already seen local ATVs in use in section

9.1 on "Simple Macros". Whenever you invoke a macro, the Assembler creates local ATVs with names P1, P2, P3 and so on, each holding one of the parameters you supplied on the macro call.

For example, the source line

```
NAME  ASET  'OSWRCH'
```

will set up an ATV called NAME, whose contents are the string 'OSWRCH'. Since the ASET directive has been used, the

ATV is global, and can be used anywhere in the assembly.

You can create local and global ATVs of the same name if you wish. However, if you wish to refer to the local ATV, you should be careful to always use the "M" form of the directives. The "A" forms will always create global ATVs, even if local ones of the same name already exist.

10.2.1.2 String and Numeric Values

All ATVs are stored by the Assembler as printable character strings. However, in many cases you will find you use them as counters for controlling loops: to make this easy, the Assembler will automatically convert ATVs from numbers to strings whenever necessary.

The rule used is quite simple. When processing ASET or MSET directives the Assembler examines the first character of the operand field. If it is a string delimiter, the operand is taken as a string and is not evaluated. If it is any other character, the Assembler treats the operand as a numeric expression, evaluates it, and converts the result into a string for storage.

Thus, the line

```
COUNT  ASET  15+3
```

will set up the ATV COUNT, containing the string "18", but

```
COUNT  ASET  '15+3'
```

sets up the string "15+3".

The operand can be any expression, provided it does not contain any forward references: thus

```
COUNT  ASET  ADDRESS+10
```

sets COUNT to hold the string form of the result of adding 10 to the address label ADDRESS.

If you use expressions that mix numeric and character literal values, some caution is needed to make sure that the Assembler does what you want: for example,

```
VALUE  ASET  'A'+3
```

would produce an error, rather than adding 3 to the code for the character 'A'. This is because the Assembler, seeing the operand

field beginning with a "'" character, will treat the operand field as a string and find that it is invalid. To overcome this, you may be able to re-order the operand expression: thus you could write the above example as

```
VALUE  ASET  3+'A'
```

Now the Assembler will treat the operand field as a numeric expression, since it does not begin with a string delimiter.

In some cases, though (particularly in macros where you may not know beforehand what the expression may look like) it may not be possible to re-order things safely. Here you can simply put a dummy number in front of the expression: thus

```
VALUE  ASET  0+'A'+'B'
```

would add the character values 'A' and 'B', the leading "0" character forcing the Assembler to treat the operand as a numeric expression.

The ALEN and MLEN directives similarly convert from number to string for you. Here the operand must be a string: for example,

```
SIZE  ALEN  'A short string'
```

sets SIZE to hold the string "14" - the length of the operand string, converted from a number into a string.

10.2.1.3 String slicing

The ASET and MSET directives also allow to you to "slice" strings, or extract characters from inside them. You perform this by adding some parameters in the operand field: for example

```
SLICE  ASET   'abcdefghij',3,2
```

will set SLICE to hold the string "cd". The second operand parameter specifies the position of the first character to be sliced out (the string starts at character 1), and the third parameter specifies the number of characters to be sliced.

If you omit the third parameter, exactly one character is sliced: thus


```
SLICE  ASET   'abcdefghij',3
```

will set SLICE to hold the string "c".

Occasionally, string manipulations such as slicing may result in strings that have leading or trailing spaces, and you may not be

able to tell beforehand how many. The ASTRIP and MSTRIP directives remove all leading and trailing spaces, so that

```
NEW   ASTRIP   '  abcd    '
```

would set the string NEW to be "abcd".

10.2.1.4 Efficient use of memory

The strings contained by ATVs are held by the Assembler in the symbol table, and so compete for memory with other symbols. You can change the contents of an ATV to a string of different length as often as you wish: however, every time the string becomes longer the Assembler will allocate a new block of memory for it. The previous block cannot be used again, so continually increasing the size of an ATV can be extremely wasteful of memory.

To overcome this, the ASTR and MSTR directives let you pre-declare a block of memory big enough to hold the maximum size string you will use. For example,

```
NAME  ASTR  50
```

sets up a block long enough to hold a 50-byte string without the need to get more memory.

The minimum amount of memory the Assembler will allocate is 5 bytes: this is enough to hold the largest possible numeric value, so that loop counters will not cause memory wastage as their values grow - you need not use ASTR and MSTR for them.

10.2.2 Simple Substitution of Assembly Time Variables

Once you have set up ATVs, you can use them to create variable source lines. We have already come across this concept in section 9.1 on "Simple Macros", in the discussion of macro parameter substitution. There, we saw that if the source of the macro contained the line

```
LDA  #@P1
```

the Assembler would replace "@P1" with whatever you had specified as

the first parameter of the macro when you called it.

ATVs are substituted into source lines in exactly the same way, and as we saw, macro parameters are in fact local ATVs.

The rule the Assembler uses is quite simple: whenever it detects an "@" sign in the source (other than in a comment line) it expects an ATV name to follow. The "@" and the name itself are replaced by the contents of the ATV before the line is assembled.

For example, you might keep the version number of a program in a small, quickly-editable file that is INCLUDEd into the source at the start of an assembly. This might contain the line

```
VERSION  ASET  "3.45"
```

which sets the ATV VERSION to hold the string "3.45". Then, at any point in the source where you needed to insert the version number (such as in a screen message) you could write, for example

```
TITLE  ASC  "SuperGame @VERSION"
```

The Assembler would replace "@VERSION" with "3.45" before it assembled the line, so the string actually planted would be

```
"SuperGame 3.45"
```

This technique lets you keep the version number in one easily-alterable place, and lets the Assembler take care of putting the up-to-date value (or parts of the value) into the source at the appropriate places.

There are some more complex uses of ATV substitution, and we shall discuss these later in section 10.3 on "Writing Complex Macros".

Some useful points to note on substitution are:

a. If you want the "@" character to appear in the line the Assembler processes, your source must contain two of them. Thus, if the line you really want to assemble is

   ```
   ASC  'VAT @ 15%'
   ```

   write it in the source file as

   ```
   ASC  'VAT @@ 15%'
   ```

b. Once the Assembler finds an "@" character in the source, it assumes that what follows is an ATV name. The end of this name is assumed to be the first non-alphanumeric character that it meets, or the end of the line. In almost all cases, this will cause no

   difficulty, but occasionally will, usually in complex macros.

   As an example, suppose you had declared at ATV called EXPR,

holding the string "10+". A subsequent source line might then read

```
LDA  #@EXPR3
```

and the intention is for this to be transformed into

```
LDA  #10+3
```

In this case, though, the substitution will fail, as the Assembler will look for an ATV called EXPR3. To force it to do as you want, write the source line as

```
LDA  #@EXPR/3
```

The "/" character enables the Assembler to detect the end of the ATV name, so it will look for EXPR as it should. The "/" will be discarded, so that the resulting line will be

```
LDA  #10+3
```

as intended. If you need a "/" character in the resulting line, write it as "//".

There is another techniques you might use in these circumstances. The Assembler does not regard spaces as significant in numeric expressions, so that you could write

```
LDA  #@EXPR 3
```

to achieve the same result.

c. No ATV substitution is performed in comment lines, in lines containing Sequence Symbols, or in the definition of macros (i.e. between the MACRO and ENDM directives). Apart from these, though, substitutions can be made at any place in a line - you can substitute for labels, opcodes, operands or any parts of them.

10.3 Writing Complex Macros

10.3.1 Programming macro loops

Mostly, you will use the Macro Programming Language to program macro loops, controlled by various conditions.

10.3.1.1 Simple loops controlled by counter

The simplest form of loop is one which is executed a fixed number of times, and needs only a counter to control it.

As an example, suppose that we need a macro to plant a number of bytes containing $FF with the DFB directive, the number being specified by the first parameter. (There are much easier ways of doing this than with a macro, of course - this only shows a general technique).

The macro definition might then be:

```
PLANT  MACRO
COUNT  MSET    0
%LOOP
       DFB     $FF
COUNT  MSET    @COUNT+1
       AIF     @COUNT<@P1,%LOOP
       ENDM
```

To see how this works, we can examine each line in turn, assuming that the macro was called with a line

```
PLANT  7
```

Line 1 : This is the macro definition line.

Line 2 : This line sets up a local ATV called COUNT, and gives it a string value of "0".

Line 3 : This is a Sequence Symbol marking the top of the loop. Note that there is nothing else on the line with it.

Line 4 : This is the DFB line that plants the $FF byte required.

Line 5 : This line increments the value of COUNT. As the Assembler reads the line, it encounters "@COUNT", which it replaces with the current string value of the ATV. Thus the first time this line is encountered, the Assembler will generate

```
COUNT  MSET  0+1
```

The second time, it generates

```
COUNT  MSET  1+1
```

and so on.

Line 6 : This tests whether the Assembler is to loop round once more. As with line 4, the Assembler will replace

"@COUNT" with the current value of the ATV. "@P1" is, of course, replaced by the first parameter of the macro. The first time round, the line processed is

```
AIF   0<7,%LOOP
```

which is true, so the Assembler skips backwards in the macro to line 4 and resumes processing from there.

10.3.1.2 Loops accessing macro parameters

Another frequently-needed form of loop is one in which all the parameters of the macro are accessed in turn. Suppose, for example, you need to write a macro THINGS, whose parameters are all numbers. Each number is to be planted in a byte in the object file with a DFB directive: to make THINGS interesting, the number of parameters must be variable.

Such a macro is fairly simple to write, but uses an ATV substitution technique that can, at first sight, be somewhat daunting. If the job were simply to plant the value of parameter 1, the line in the macro that does it would simply be

```
DFB   @P1
```

However, we need to access each parameter in turn: the Assembler must somehow be made to see "@P1" the first time round the loop, "@P2" in the second, and so on. Effectively, then, we need a substitution technique that lets us have a variable ATV name: i.e. one that first substitutes the number ("1", "2", "3" etc) then substitutes for the ATV name so formed.

The Assembler can do this easily, since ATV substitution operates in a hierarchic fashion. For example, suppose that a source line contains

```
@(P@COUNT)
```

somewhere.

On seeing the first "@" character, the Assembler prepares to substitute an ATV. It finds, though, that the next character is a "(", so that it now expects a bracketed expression rather than an ATV name. It notes where it has got to in the source line, and explores within the brackets.

Now, though, it stores the characters it finds (rather than passing them on to be assembled), and expects to end up with a valid ATV name by the time it gets to the ")" character. It notes the "P", then finds the second "@". This makes it try again to

substitute an ATV, and this time it finds a real ATV called COUNT, which we shall suppose contains the string "1". After "COUNT" it finds the ")" ending the bracketed expression; thus, within the brackets the string it has built is now "P1".

Having ended the bracketed expression, the Assembler goes back to where it was.The "(P@COUNT)" has provided the string "P1", and this now is a valid ATV name. So it proceeds to substitute the value of ATV P1, the first macro parameter, as we intended.

To see how we might use this technique, we can consider a definition of THINGS:

```
THINGS  MACRO
COUNT   MSET    1
%LOOP
        AIF     '@(P@COUNT)'='',%ALL.DONE
        DFB     @(P@COUNT)
COUNT   MSET    @COUNT+1
        AIF     @COUNT<10,%LOOP
%ALL.DONE
        ENDM
```

To see how this works, we can examine each line in turn.

Line 1 : This is the macro definition line.

Line 2 : This line sets up a local ATV called COUNT, and gives it a string value of "1".

Line 3 : This is a Sequence Symbol marking the top of the loop.

Line 4 : Since the number of parameters must be variable, we need to test whether we've processed them all. You can see the substitution technique discussed above in use here to check if the next parameter is null - any parameters that you don't supply in a macro call are strings of zero size. If the parameter is null, the Assembler skips forwards in the macro until it gets to the Sequence Symbol %ALL.DONE.

Line 5 : This is the DFB line that plants the current macro parameter as a byte.

Line 6 : This line increments the value of COUNT, as in the previous example.

Line 7 : This tests whether the Assembler is to loop round once more. The final macro parameter is P9, so once COUNT reaches 10 the macro is finished.

Line 8 : This is the Sequence Symbol that line 4 skips to if it

finds a null parameter.

Thus, if the macro was called with a line

```
THINGS  1,$FE,FRED-1
```

the loop would be traversed three times, and the lines actually assembled would be

```
DFB     1
DFB     $FE
DFB     FRED-1
```

The technique of hierarchical substitution, though most often used to access macro parameters in turn, can be used in many other applications: you can nest substitutions to as deep as you are likely to need, so that you might write something as horrendous looking as

```
LDA   #@(XX@(COUNT@PTR)B)
```

if you really needed to.

10.3.2 Changing macro parameters

Macro parameters are in fact local ATVs with names P1, P2, P3 and so on. This means that you can change them within a macro as you wish.

One example of this might be to allow the use of default macro parameters (although section 10.3.3 below shows an automatic way to do this). Suppose that a macro parameter should be a number, whose default value is 1. You could define it as:

```
TEST  MACRO
      AIF    '@P1'#'',%NEXT
P1    MSET   1
%NEXT
      LDA    #@P1
      ENDM
```

Within this example, we have:

Line 1 : The macro definition line.

Line 2 : This tests if parameter 1 has been supplied. If so, it will not be a null string, so the Assembler skips to %NEXT.

Line 3 : This line sets parameter 1 to the default value.

Line 4 : This is the Sequence Symbol skipped to if the parameter is not defaulted.

Line 5 : This line actually uses the parameter. It will assemble correctly even if the parameter was not given, since in that case line 3 will have set it up to be the default value.

10.3.3 Setting default macro parameters

The example above showed one way of establishing a default value of a macro parameter, but in fact the Assembler gives you an automatic and easy way of doing this with the DEFPARS directive.

The effect of this directive, which you can issue at any time inside a macro, is to set the values of any of the macro parameters P1, P2, P3 and so on, unless they are already defined.

For example, you might call a macro with a line

```
FRED    1,,3
```

where you have defined parameters 1 and 3, but not parameter 2. If the macro now executes, say,

```
DEFPARS  100,200,300
```

the Assembler will check the parameters in turn. Parameters 1 and 3 are already defined, so the "100" and "300" values are not used. Parameter 2, though, is not yet defined, so it is set to "200" from this point on.

If you wished, say, to establish a default for only some of the parameters of a macro, simply specify only those parameters in the DEFPARS directive and default the others. Thus

```
DEFPARS ,,,44
```

sets a default for parameter 4, but has no effect whatsoever on parameters 1, 2, 3, 5, 6, 7, 8 and 9.

10.3.4 Listing control for macros

If you write complex macros with lots of loops, you will find

that the Assembler actually executes a large number of lines that just contain the AIF, MSET directives, etc. This can swamp the

listing, and make it hard to see the actual directives that plant data or code (as well as using up a large amount of paper).

To overcome this, list level 2 will not list directives such as MSET, AIF, etc, unless they contain errors. In order to see all the lines of a macro expansion, use list level 3. (Note that outside macros, ASET and other similar directives will list at level 2).

10.3.5 Exiting a macro prematurely

You may sometimes need to exit a macro as a result of a test. Depending on circumstances, you may be able to use AIF or AGO to skip to the physical end of the macro; however, you can also use the MEXIT directive. This exits the macro immediately, wherever in the macro body it is encountered.

10.3.6 Comment lines within macros

You can include comment lines within the definitions of macros, just as in normal code. For example, a short macro might be defined as

```
TEST   MACRO
*
* Adjust value in PTR0
*
       INC    PTR0
       ENDM
```

The three comment lines will be stored in the macro definition, and (if the listing level is set to 2 or higher) will be listed whenever you call the macro. This lets you output comments to describe the actions of the macro.

The disadvantage of this technique is that the comment lines are stored in their entirety in memory when the macro is defined, giving you correspondingly less space for the symbol table.

When you come to writing complex macros making great use of ATV's and loops, you will find it most advantageous to include liberal amounts of comment. In order that these comments, which purely document how the macro works rather than the code it generates, shouldn't use memory, comment lines beginning with "!" are not stored. Thus, you could write a macro as

```
TEST  MACRO
!
! See if first parameter is null
!
```



```
      AIF    '@P1'#'',%NEXT
!
! It is null, so reset it to "1"
!
P1    MSET   1
%NEXT
!
! Parameter 1 now definitely has a value,
! so we can use it
!
      LDA    #@P1
      ENDM
```

The lines beginning "!" will appear in the listing when you define the macro, but will not be stored and will not appear when the macro is called.

10.4 System ATVs

The Assembler provides a number of read-only ATVs that you can substitute. They provide information on the Assembler and the environment that you can use to control assembly. Each system ATV name starts with a "$" character: they are

$CLST	The current code listing level.
$DEFCLST	The default code listing level.
$DEFLST	The default listing level.
$FLEVEL	"1" if the current file is an INCLUDE file; "0" if it is not.
$FS	The number of the filing system in use, as returned by OSARGS with A=0, Y=0. The Acorn DFS returns "4", Econet returns "5" and ADFS returns "8".
$LINE	The number of the current source line.
$LST	The current listing level.
$MC	The macro call counter, used to generate unique labels within macros (see section 9.5)

$MLEVEL The current macro nesting level. If not in a macro, the value is "0".

$OBJECT The name of the current object file, which may include leading or trailing spaces.

$OS The version of the MOS in use, as returned by OSBYTE with A=0.

$OSHWM The primary OSHWM value of the machine running the Assembler.

$PLENGTH The current page depth used in the assembly listing, as set by the PAGE directive.

$PWIDTH The current line width of the asembly listing, as set by the PAGE directive.

$PRINTING Indicates whether the assembly listing is being sent to the screen or to a printer or file. The value returned is "0" if the listing is being sent to the screen, and "-1" if it is being sent to a printer or a file.

$SOURCE The name of the current source file, which may include leading or trailing spaces.

$TIME The currently-set timestamp string, which may include leading or trailing spaces.

$TTL The currently-set page title string, which may include leading or trailing spaces.

$VERSION The version of the Assembler in use. This is returned as a numeric string, so that version 1.60 sets the string to be "160".

For example, the line

```
ORG  @$OSHWM
```

could be used to set the base address of the code to the OSHWM value of the machine being used, without the need to know what that value was.

Appendix 1 : OPCODES AND ADDRESSING MODES

The Assembler supports all the opcodes of the 6502 and 65C02 processor families, using standard MOSTEK mnemonics. For descriptions of the opcodes, see for example "The Advanced User Guide for the BBC Micro" (for 6502-compatible opcodes) and the "Master Reference Manual Part 2" (both 6502- and 65C02-compatible opcodes, although it describes the BBC BASIC Assembler syntax which cannot be used with this Assembler.)

The 6502-compatible opcode mnemonics available are:

```
ADC  AND  ASL  BCC  BCS  BEQ  BIT  BMI  BNE  BPL  BRK  BVC  BVS  CLC
CLD  CLI  CLV  CMP  CPX  CPY  DEC  DEX  DEY  EOR  INC  INX  INY  JMP
JSR  LDA  LDX  LDY  LSR  NOP  ORA  PHA  PHP  PLA  PLP  ROL  ROR  RTI
RTS  SBC  SEC  SED  SEI  STA  STX  STY  TAX  TAY  TSX  TXS  TYA
```

The 65C02-only mnemonics are:

```
BRA  DEA  INA  PHX  PHY  PLX  PLY  STZ  TRB  TSB
```

The opcode CLR can be used as a synonym for STZ.

The addressing modes common to both the 6502 and 65C02 processors are:

Mode	Syntax
Implied	op
Accumulator	op A or op
Immediate	op #expr8
Low byte	op #>expr
High byte	op #<expr
Zero page	op expr8
Indexed by X	op expr8,X
Indexed by Y	op expr8,Y
Absolute	op expr16
Indexed by X	op expr16,X
Indexed by Y	op expr16,Y
Indirect pre-indexed	op (expr8,X)
Indirect post-indexed	op (expr8),Y
Absolute indirect	op (expr)

The addressing modes usable with the 65C02 processor only are:

Mode	Syntax
Zero-page indexed	op (expr8)
Absolute indirect pre-indexed	op (expr16,X)

In these definitions, "expr8" represents an 8 bit number; "expr16"

a 16-bit number; and "expr" a value that is either 8 or 16 bits long.

Appendix 2: ASSEMBLER DIRECTIVES

The Assembler supports a large number of directives, or pseudo-ops. This section gives a detailed definition of each, arranged in alphabetical order.

Directives follow the normal syntax of opcodes. All can be followed by comment fields started with a ";" character; however, not all directives may have labels.

In the specification, items enclosed in "[]" brackets are optional, and can be omitted.

AGO Skips to a Sequence Symbol.

Syntax:

```
AGO  <sequence>
```

Example:

```
AGO  %LOOP
```

AIF Skips to a Sequence Symbol if a condition is true.

Syntax:

```
AIF  <expr>,<sequence>
```

or

```
AIF  <stringexpr>,<sequence>
```

If <condition> is false assembly continues in the next line.

Example:

```
AIF  'FREDA'>'FRED',%LOOP
```

AIFDEF Skips to a Sequence Symbol if a symbol has been defined.

Syntax:

```
AIFDEF  <symbol>,<sequence>
```

Example:

```
AIFDEF  RAM.CODE,%DOING.RAM
```

AIFNDEF Skips to a Sequence Symbol if a symbol has not been defined.

The syntax is as for AIFDEF.

ALEN Sets a global ATV to the length of the operand string.

Syntax:

```
<label>  ALEN  <string>
```

Example:

```
SIZE  ALEN  'This is a string'
```


ASC Defines an ASCII string.

Syntax:

```
[<label>]  ASC  <string>
```

Within <string> you can use the techniques of section 4.3.3 to plant control characters or characters with bit 7 set. If a label is specified, it is set to the address of the first byte of the string.

Example:

```
TITLE  ASC  'SuperGame 3.45'
```

ASET Sets a global ATV to contain a string.

Syntax:

```
  <label>  ASET  <string>[,<expr1>[,<expr2>]]
or
  <label>  ASET  <expr>
```

The final two parameters in the operand field allow the string parameter to be sliced. Both are expressions that must not involve forward references. <expr1> specifies the first byte to be sliced from the string (the first byte in the string is byte 1). <expr2> defines the size of the slice; if omitted, one byte is extracted.

Example:

```
COUNT  ASET  100
```

ASTR Allocates space for a global ATV.

Syntax:

<label> ASTR <expr>

<expr>, which must not contain forward references, defines the space needed. The value can be up to 255 bytes.

Example:

NAME ASTR 150

ASTRIP Sets a global ATV to a string value, removing leading and trailing spaces in the process.

Syntax:

<label> ASTRIP <string>

Example:

MIDDLE ASTRIP '   aardvark   '

CASC This is similar in action to ASC, but will plant a byte containing the number of characters within the string before the string itself.

CHN Closes the current source file and starts another.

Syntax:

[<label>] CHN <filename>

The directive is not allowed in an INCLUDE file.

Example:

CHN :2.NEXTBIT

CHR Defines the character used by the REP directive.

Syntax:

[<label>] CHR character

The character is specified literally, and not as a character constant.

Example:

```
CHR  #
```

CLST Sets the code listing level.

Syntax:

```
[<label>]  CLST  [<expr>]
```

<expr>, which must not contain forward references, gives the new listing level. If omitted, the level is reset to the default from the command line. The values allowed are:

0 List only the first 3 bytes produced by any line.

1 List all the bytes planted by any line except a CODE directive.

2 List all the bytes planted by all lines.

Example:

```
CLST  2
```

CODE Includes a file (such as a screen dump) directly into the object file.

Syntax:

```
[<label>]  CODE  <filename>
```

The specified file is copied, with no interpretation, directly to the object file. If <label> is specified it is set to the address of the first byte copied. The address count is increased by the size of the file.

Example:

```
CODE  SCRDUMP
```

CSTR Similar to STR, but this directive plants a byte containing the number of characters in the string before the string itself. The count includes the $0D byte that is automatically added at the end of the string.

DB This is identical to DFB.

DDB Defines a number of two-byte values, output in high-low order.

Syntax:

```
[<label>]  DDB  <expr>[,<expr>...]
```

Each expression in the operand field is evaluated and stored as a two-byte value. If <label> is present, it is set to the address of the high-order byte of the first expression.

Each expression can be preceded by a repeat count as described in section 4.2.3.

Example:

```
TABLE  DDB  28,32105,[10]$FFFF
```


DEFPARS Sets up default parameters for a macro. See section 10.3.3 for full details of this directive.

DEND Terminates a DSECT.

Syntax:

```
[<label>]  DEND
```

The address that was saved when the DSECT directive was encountered is reset.

Example:

```
DEND
```

DFB Defines a number of single-byte values.

Syntax:

```
[<label>]  DFB  <expr>[,<expr>...]
```

Each expression in the operand field is evaluated and stored as a byte value. If <label> is present, it is set to the address of the first expression.

Each expression can be preceded by a repeat count as described in section 4.2.3.

Example:

```
TABLE  DFB  0,[27]1,FRED+2
```

DFDB This is identical to DDB.

DFS This is identical to DS.

DFW This is identical to DW.

DISP Displays a message on both pass 1 and pass 2.

Syntax:

```
[<label>]  DISP  <string>
```

<string> may contain references to variables: for details see section 8.1.

Example:


```
DISP  'Program size is %D(*-$8000)'
```

DISP1 As for DISP, except that the message is displayed only on pass 1.

DISP2 As for DISP, except that the message is displayed only on pass 2.

DO A synonym for IF.

DS Reserves space in the object file, setting the bytes so affected to zero.

Syntax:

```
[<label>]  DS  <expr>
```

<expr>, which must not include forward references, defines the amount of space to be reserved. If <label> is present it is set to the address of the first byte reserved.

Example:

```
                    DS  $C000-PROG.TOP.ADDRESS
```

DSECT Opens a "dummy section" that allows an area of memory to be defined without generating any object code.

Syntax:

```
                  [<label>]  DSECT
```

When the DSECT directive is processed, the current address is noted by the Assembler, and the value is reset to that at the end of the previous dummy section (or 0, if this is the first). An ORG directive can be used to change the value if required.

When the dummy section is terminated by a DEND directive the old value of the address is restored.

Example:

```
                        DSECT
```

DW Defines a number of two-byte values output in low-high order.

Syntax:

```
          [<label>]  DW  <expr>[,<expr>...]
```

Each expression in the operand field is evaluated and stored as a two-byte value. If <label> is present, it is set to the address of the low-order byte of the first expression.

Each expression can be preceded by a repeat count as described in section 4.2.3.

Example:

```
                TABLE  DW FRED,BERT
```

ELSE Marks the end of the TRUE branch of a conditional assembly block begun by IF, IFDEF and IFNDEF. For details see section 7.

ENDM Terminates the definition of a macro begun by the MACRO directive.

Syntax:

ENDM

EQU Assigns a value to a symbol.

Syntax:

<label> EQU <expr>

<expr> must not contain forward references.

Example:

OSWRCH EQU $FFEE

EXEC Specifies the bottom 16 bits of the 32 bit execution address of the object code.

Syntax:

[<label>] EXEC <expr>

The catalogue entry of the object file will be updated to show the specified execution address. <expr> can only be a 16-bit quantity: the value of the two top bytes of the address are set by the MSW directive.

If this directive is not used, the execution address is set to the address defined by the first ORG that is not in a dummy section.

Example:

EXEC START+3

FI Terminates a conditional assembly block begun by IF, IFDEF or IFNDEF. For details, see section 7.

FIN This is identical to FI.

HEX Defines a series of bytes in the object program, the bytes being specified as a hexadecimal string.

Syntax:

[<label>] HEX <string>

<string> must contain hexadecimal digits (i.e. '0'..'9' or 'A'..'F'), and each pair of digits is output as one byte.

If <label> is present it is set to the address of the first byte from the string.

Example:

```
BYTE.LIST  HEX  'AB34FF2E7E'
```

IF Begins a conditional assembly block.

Syntax:

```
[<label>]  IF  <expr>
```

If <expr>, which cannot contain forward references, is non-zero, the condition is TRUE, otherwise it is FALSE. For details, see section 7.1.

IFDEF Begins a conditional assembly block.

Syntax:

```
[<label>]  IFDEF  <symbol>
```

If <symbol> has been defined, the condition is TRUE, otherwise it is FALSE. For details, see section 7.2

IFNDEF Begins a conditional assembly block.

Syntax:

```
[<label>]  IFNDEF  <symbol>
```

If <symbol> has not been defined, the condition is TRUE, otherwise it is FALSE. For details, see section 7.2.

INCLUDE Begins assembly from a new source file, afterwards resuming with the current file.

Syntax:

```
[<label>]  INCLUDE  <filename>
```

The Assembler will process lines from the specified file. Lines in the listing file will begin with an "I" to indicate their origin. At end-of-file, it resumes after the INCLUDE

directive.

The directive may not be used within an INCLUDE file or a macro.

Example:

```
INCLUDE  :3.DEFINES
```

INFO This is identical to DISP2.

LFCOND Specifies that lines skipped in a conditional are to be listed.

Syntax:

```
[<label>]  LFCOND
```

For details, see section 7.4.

LOAD Specifies the bottom 16 bits of the 32 bit load address of the object code.

Syntax:

```
[<label>]  LOAD  <expr>
```

The catalogue entry of the object file will be updated to show the specified load address. <expr> can only be a 16-bit

quantity: the value of the two top bytes of the address are set by the MSW directive.

If this directive is not used, the load address is set to the address defined by the first ORG that is not in a dummy section.

Example:

```
LOAD  $1902
```

LST Sets the source listing level.

Syntax:

```
[<label>]  LST  [<expr>]
```

<expr>, which must not contain forward references, give the new level. If omitted, the level is reset to the default

specified in the command line.

The allowed values are:

0 No lines are listed.

1 List lines from files, but not from macros.

2 List all lines, but not directives such as AIF within macros.

3 List all lines.

Lines containing errors are always listed.

MACRO Defines a macro.

Syntax:

<label> MACRO

The following lines are stored as the definition of the macro. No ATV substitution is performed on the lines. The definition is terminated by an ENDM directive. Within the definition, comment lines starting with "!" will be listed but not stored in memory.

The macro name may not be more than 8 characters long.

The MACRO directive may not be used within a macro.

Example:


```
THING  MACRO
```

MEXIT Causes a macro to terminate immediately.

Syntax:

MEXIT

MLEN As for ALEN, but this can be used only within a macro and creates a local ATV rather than a global one.

MSET As for ASET, but this can be used only within a macro and creates a local ATV rather than a global one.

MSTR As for ASTR, but this can be used only within a macro and creates a local ATV rather than a global one.

MSTRIP As for ASTRIP, but this can be used only within a macro and creates a local ATV rather than a global one.

MSW Sets the two top bytes of the load and execution addresses.

Syntax:

[<label>] MSW <expr>

By default the bytes used are $FFFF, but if you are assembling code to run on a second processor you will need to change them to the appropriate value.

Example:

```
MSW  0
```

ORG Sets the value of the address counter.

Syntax:

[<label>] ORG <expr>

<expr>, which must not contain forward references, gives the new value. The first ORG directive that is not in a DSECT..DEND block also defines the default load and execution addresses.

The object file is not affected in any way; thus if you use ORG to advance the address value, the Assembler will not plant

any bytes in the object file to fill the gap. To accomplish this effect, you can use the DS directive.

Example:

```
ORG  $1900
```

PAGE Defines the page length and width of the listing.

Syntax:

[<label>] PAGE <expr1>[,<expr2>]

<expr1> is the physical number of lines on the paper (default 66). The Assembler leaves a small gap at the end of each page,

to avoid the perforations.

<expr2> is the number of characters to print on each line. If omitted, the value is set to 80.

PAGE has no effect on the screen display.

Example:

```
                    PAGE  88,132
```

QUERY Displays a question and reads in a reply, which is evaluated as an expression and used to define a label.

Syntax:

```
               <label>  QUERY  <string>
```

For details, see section 8.3

REP Outputs a comment line to the listing to mark a source division.

Syntax:

```
               [<label>]  REP  [<expr>]
```

<expr> must not exceed 132 and must not contain forward references. If you omit the value, the Assembler will fill the output line to its full width.

For details, see section 3.5.4

SFCOND Specifies that lines skipped in a conditional should not be listed.

Syntax:

```
                  [<label>]  SFCOND
```

For details, see section 7.4

SKP Leaves a gap in the listing or starts a new page.

Syntax:

```
                [<label>]  SKP  <expr>
   or
                [<label>]  SKP  H
```

In the first form, <expr> gives the number of lines to be left blank. In the second, a new page is started.

The SKP directive itself is not listed.

Example:

```
                        SKP  5
```

STOP Aborts the assembly on pass 1, outputting a message.

Syntax:

```
              [<label>]  STOP  <string>
```

The Assembler will display <string>, then abort with the error message "Stopped".

Example:

```
            STOP  'Too much in page zero'
```

STR This is similar to ASC, except that the Assembler will automatically add a $0D byte (Carriage Return) to the bytes generated.

SYSCLI Issues a MOS command, allowing you, for example, to change the default disc drive.

Syntax:

```
             [<label>]  SYSCLI  <string>
```


You need not begin the string to be issued as a MOS command with a "*", but it doesn't matter if you do. Note that care is needed over the commands you try to execute. Anything that uses memory will almost certainly corrupt the Assembler's workspace and cause assembly to go horribly wrong.

Example:

```
                  SYSCLI  'DRIVE 2'
```

SYSVDU Outputs one or more bytes to the screen and printer.

Syntax:

[<label>] SYSVDU <expr>[,<expr>...]

The directive is similar to the BASIC VDU command. The low-byte of each <expr> is written with a call to OSWRCH. This gives you a means of sending control codes to a printer to set page formats, etc.

SYSVDU will send all the bytes you specify to the screen. If you have used the "-P" command-line option, they will also be sent to the printer. To send the bytes only to the printer, precede each with a "1" value. The bytes are not sent to a listing file selected with the "-F" option.

SYSVDU operates on both passes of the Assembler.

Example:

```
SYSVDU  1,27,1,'E'
```

SYSVDU1 This directive is identical to SYSVDU, but operates only on pass 1.

SYSVDU2 This directive is identical to SYSVDU, but operates only on pass 2.

TABS This directive allows you to redefine the TAB positions the Assembler will use when listing source lines.

Syntax:

[<label>] TABS <expr>[,<expr>...]

You may specify up to 14 TAB positions: any excess values will be ignored. Specifying the directive with no operand filed will reset the built-in defaults.

See section 3.2 for more details.

TTL Defines the title string on the top of each listing page.

Syntax:

[<label>] TTL <string>

The Assembler uses the first 20 bytes of the string as the page title, and also starts a new page.

Example:

```
TTL  'Screen Dumper'
```

TIME Defines the timestamp string output on each listing page.

Syntax:

```
[<label>]  TIME  <string>
```

The Assembler uses the first 24 bytes of the string for the new timestamp. The value need not be a time; you can use it as a subsidiary page title.

On a Master 128 the directive is ignored, as the timestamp is read from the real-time clock. The Assembler can also be configured to read a third-party real-time clock fitted to other models.

Example:

```
TIME  '25-August-1986 13:00'
```

WAIT This displays a string (see under DISP), then pauses the assembly until you press a key.

WAIT1 As for WAIT, but operates only on pass 1.

WAIT2 As for WAIT, but operates only on pass 2.

Appendix 3. DIFFERENCES FROM THE ADE ASSEMBLER

The 65C02 Assembler contains a number of similar directives to the ADE Assembler, and in almost all cases it will be possible to assemble programs written for ADE with it.

A few directives, though, have some minor differences, and there are also some small points on source format that must be considered.

- Only single- and double-quote characters may delimit strings.

- The second delimiter must not be omitted from character constants.

- Character constants obey the same rule as strings in the ASC and STR directives for control characters and those with bit 7 set. Thus the "|" and "^" characters need to be doubled if they are being defined as constants.

- Strings containing commas or leading or trailing spaces are specified as macro parameters within normal string delimiters rather than in [] brackets.

- Macro parameters must always be separated by commas.

- The SYSFX directive is ignored: to achieve the desired effect you should use the SYSCLI directive to issue a *FX command.

- Macro libraries are not implemented.

- Local labels within macros are not implemented. The system ATV $MC should be used to construct unique labels as described in section 9.5.

- Expressions are evaluated fully hierarchically rather than left-to-right.

- The "#" arithmetic operator is used for "inequality", returning -1 if unequal and 0 if equal, rather than "modulus".

- The LST directive has a numeric operand rather than the strings OFF, ON and FULL. For ADE compatibility, define symbols OFF, ON and FULL with the EQU directive to have values 0, 1 and 2. List level 3 has no equivalent in ADE.

- The load and execution addresses are set by the first ORG directive only. To set them to different values use the LOAD and EXEC directives.

Appendix 4. DIFFERENCES FROM PREVIOUS RELEASES

Version 1.60 of the 65C02 Assembler includes several new facilities and a number of corrections from the previous 1.52 release. The most important changes made are as follows:

- A new directive, TABS, allows the layout of listing lines to be configured. A patch facility allows the default settings to be configured.

- New directives SYSVDU1 and SYSVDU2 perform the same action as SYSVDU but only on pass 1 and pass 2 respectively.

- Comment lines can be started with a "!" character; such lines within macro definitions are not stored in memory and can be used to annotate the macro itself.

- Printer handling has been extensively rewritten to avoid problems when using Econet print servers.

- Assembly listings can be directed to a file with the "-F" command-line option.

- String slicing operations in the ASET and MSET directives now work exactly as documented. In version 1.52, the string start and slice size parameters were reversed, and the first byte of the string was numbered 0 and not 1.

- A new MOS command, *CLI, allows the Assembler's CLI mode to be explicitly entered from any language.

- A REP directive with no operand now fills the output line to its defined width.

- System ATVs $PWIDTH and $PLENGTH allow the values set by the PAGE directive to be read.

- System ATV $PRINTING allows the source to determine if an output listing is being sent to printer or file, or to the screen only.

- A repeat count of zero in directives such as DB causes no bytes to be written to the object file, rather than repeating the operand 65536 times.

- A zero operand in the DS directive causes a gap size of zero rather than 65536.

- Patch facilities allow the Assembler's translation of source bytes in the range 128..255 to be configured.

- Patch facilities force the Assembler to read a real-time clock on machines other than a Master 128.

- Patch facilities enable the "~" character to be used in character constants and strings to plant the following character as a control character with bit 7 set.

- Patch facilities allow the Form Feeds sent to a printer at the end of a listing to be suppressed.

- Source file repositioning (in directives such as AGO, AIF) no longer causes random failures with files larger than 14kbytes on a second processor.

- The DB directive now accepts negative operands in the range -1 to -127.

Appendix 5. CUSTOMISING THE ASSEMBLER

Version 1.60 of the Assembler contains a number of options that can be configured or activated by changes to the ROM image. Typically the facilities controlled in this way are highly specific to particular hardware or modes of working, rather than of general use.

In order to configure the Assembler you will need to alter some bytes within it, using a machine code monitor such as Exmon or Beebmon. The things you need to do a simple and straightforward, but as with all such operations, you should make sure you have an unaltered copy of the Assembler stored safely on a disc before you begin. You will need to be careful that you alter the correct locations and put in the correct values: the Assembler will be unlikely to work correctly, and probably will not work at all, if you make a mistake.

A5.1 The patch area

In order to patch the Assembler, you will first need to locate the patch table. This is an area of memory that either contains the actual locations you alter, or contains the addresses of the locations in other parts of the ROM.

The patch area is not held at a fixed absolute address, but will always follow the standard sideways ROM header. To find it, look first at byte $8007 in the ROM (in this section, all addresses are given from the base of $8000, which is the memory address the Assembler runs at. To get the offset within a file for the bytes concerned, simply subtract $8000 from all values). Add the value of byte $8007 to $8000, to obtain the address of the copyright string (in version 1.60 it reads "(C)1987 Alan Phillips").

At the end of the copyright string you will find a byte containing the value of zero; the patch area starts at the byte immediately following this one.

Athough the absolute position of the patch area is not fixed between releases, the format of it is, so that the following sections apply to 1.60 and any future releases.

A5.1.1 The default TAB position list

Bytes 0 and 1 of the patch table are the address of the default TAB table used by the Assembler when listing source lines. (Note that, as usual, the address is stored with the least significant byte first).

The address in the patch table takes you to the area of the ROM where the default TAB positions are held. To change them, you should alter the first byte of the table to contain the number of TAB positions you wish to define (the number you use must be between 1 and 14). Following this you should place the TAB

positions themselves; they must be in ascending numeric order.

For example, to define 5 default TAB positions at columns 8, 17, 25, 32 and 40, you would change the bytes in the TAB table to be

```
5  8  17  25  32  40
```

A5.1.2 Source character translation

Byte 2 of the patch area controls how the Assembler treats characters in the source file with bit 7 set. Normally the value is zero, instructing the Assembler to ignore such bytes. Changing it to some other value makes the Assembler replace such bytes with the value you supply.

Thus, for example, you could change the byte in the patch table to the value $40 (the code for a space character) to replace source characters with bit 7 set with a space).

It is up to you to make sure that the value you use is sensible: you should set the byte either to a space or to a TAB character.

A5.1.3 Real time clock handling

Byte 3 of the patch area controls how the Assembler uses any real-time clock you may have fitted. By default the value is $FF, and the Assembler will attempt to read a real-time clock only on a Master 128. You may, though, have fitted a third-party clock to another model: setting the byte value to 0 will make the Assembler try to read the time on all models.

Note, though, that your real-time clock must respond to exactly the same *FX calls as used on the Master 128: the Assembler has no means of knowing how to read the time from clocks that are operated in other ways.

A5.1.4 Character constant and string options

Byte 4 of the patch area controls how the "~" character is treated in character constants and strings. By default the value of the byte is zero, and the character has no special effect. Changing the value to $FF causes the "~" character to take on special meaning: it will now cause the following character to be planted as a control character with bit 7 set. Naturally, you would need to write '~~' in a character constant to obtain the "~" character itself if you turn this option on.

A5.1.5 Page throws at the end of a listing

Byte 5 of the patch area controls whether the Assembler outputs Form Feed characters to a printer or listing file at the end of a listing. By default the value of the byte is $FF: this makes the Assembler output a Form Feed both before and after the lines of statistics that tell you the number of errors, etc. Changing the byte to 0 suppresses the Form Feeds, so the statistics immediately follow the symbol table listing, with no throw to head-of-form following them.